III[rd] Year B.Tech CSE-I Sem

SOFTWARE ENGINEERING

Syllabus:

UNIT I :

Introduction to Software Engineering : The evolving role of software, Changing Nature of Software, Software myths.

A Generic view of process : Software engineering- A layered technology, a process framework, The Capability Maturity Model Integration (CMMI), Process patterns, process assessment, personal and team process models

Process models : The waterfall model, Incremental process models, Evolutionary process models, The Unified process.

UNIT II :

Software Requirements : Functional and non-functional requirements, User requirements, System requirements, Interface specification, the software requirements document**.**

Requirements engineering process : Feasibility studies, Requirements elicitation and analysis, Requirements validation, Requirements management.

System models : Context Models, Behavioral models, Data models, Object models, structured methods.

UNIT III:

Design Engineering : Design process and Design quality, Design concepts, the design model.

Creating an architectural design : Software architecture, Data design, Architectural styles and patterns, Architectural Design.

Object-Oriented Design : Objects and object classes, An Object-Oriented design process, Design evolution.

Performing User interface design : Golden rules, User interface analysis and design, interface analysis, interface design steps, Design evaluation.

UNIT IV : Testing Strategies : A strategic approach to software testing, test strategies for conventional software, Black-Box and White-Box testing, Validation testing, System testing, the art of Debugging.
Product metrics : Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Metrics for Process and Products : Software Measurement, Metrics for software quality.

UNIT IV :
Risk management : Reactive vs. Proactive Risk strategies, software risks, Risk identification, Risk projection, Risk refinement, RMMM, RMMM Plan.

Quality Management : Quality concepts, Software quality assurance, Software Reviews, Formal technical Reviews, Statistical Software quality Assurance, Software reliability, The ISO 9000 quality standards.

TEXT BOOKS :

1. Software Engineering, A practitioner's Approach- Roger S. Pressman, 6th edition.McGrawHill International Edition.
2. Software Engineering- Sommerville, 7th edition, Pearson education.

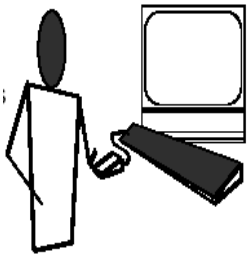
REFERENCES :

1. Software Engineering- K.K. Agarwal & Yogesh Singh, New Age International Publishers

2. Software Engineering, an Engineering approach- James F. Peters, Witold Pedrycz, John Wiely.
3. Systems Analysis and Design- Shely Cashman Rosenblatt,Thomson Publications.
4. Software Engineering principles and practice- Waman S Jawadekar, The McGraw-Hill.

1. INTRODUCTION TO SOFTWARE ENGINEERING

- **What is Software?**
  - Computer Software is the product that software professional design and built. It includes
    - Programs
    - Content
    - Documents



- **What is software engineering?**
  - *Your thoughts here*
    - Related to the process: a systematic procedure used for the analysis, design, implementation, test and maintenance of software.
    - Related to the product: the software should be efficient, reliable, usable, modifiable, portable, testable, reusable, maintainable, interoperable, and correct.
  - *The definition in IEEE Standard*:
    - The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software.
    - The study of approaches as in 1993: The Joint IEEE Computer Society and ACM Steering Committee for the establishment of software engineering as a profession.

- **What is important and what are the steps?**
  - Software affects nearly every aspect of our lives. (data).
  - You build software like you build any successful product, by applying a process that leads to a high-quality result.
  - You apply a software engineering approach.

- **What is the work Product?**
  - From the point of view of a software engineer, the work product is the programs, documents, and content.
  - From the user's viewpoint, the work product is the resultant information that somehow makes the user's world better.

- *The Product*

  - Is the engine that drives business decision making.
  - Software serves as the basis for modern scientific investigation and engineering problem solving.
  - It is embedded in systems of all kinds: transportation, medical, telecommunications, military, industrial processes, entertainment, office products…
  -

1.1 THE EVOLVING ROLE OF SOFTWARE

■ **Software's Dual Role**

- *Software is a product*
  - Delivers computing potential
  - Produces, manages, acquires, modifies, displays, or transmits information.

- *Software is a vehicle for delivering a product*
  - Supports or directly provides system functionality
  - Controls other programs (e.g., an operating system)
  - Effects communications (e.g., networking software)
  - Helps build other software (e.g., software tools)

- The Law of Continuing Change (1974): E-type systems must be continually adapted else they become progressively less satisfactory.

- The Law of Increasing Complexity (1974): As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

- The Law of Self Regulation (1974): The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

- The Law of Conservation of Organizational Stability (1980): The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

- The Law of Conservation of Familiarity (1980): As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution.

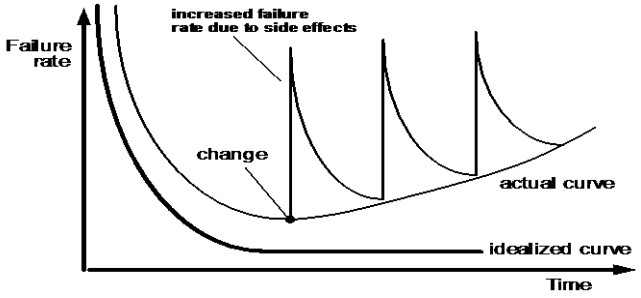
- The Law of Continuing Growth (1980): The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

- The Law of Declining Quality (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

- The Feedback System Law (1996): E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

1.2 SOFTWARE CHARACTERISTICS

- *Software is developed or engineered; it is not manufactured in the classical sense*.
- *Software does not wear out.*



Failure curve for software (idealized and actual curves)

- *Most software is custom-built, rather than being assembled from existing components.*

■ Software Components

- In the hardware world, component reuse is a natural part of the engineering process.
- In the software world, it is something that has yet to be achieved on a broad scale.
- Reusability is an important characteristic of a high-quality software component.
- A software component should be designed and implemented so that it can be reused in many different programs.
- In the 1960s, we built scientific subroutine libraries that wear reusable in a broad array of engineering and scientific application.
- These subroutine libraries reused well defined algorithms in an effective manner, but had a limited domain of application.
- Today, we have extended our view of reuse to encompass not only algorithms, but also data structures.

1.3 THE CHANGING NATURE OF SOFTWARE

Software Applications

- System software: system software is collection of programs written to service other programs. E.g. Compilers, editors, file management utilities, operating systems, drivers, networking etc.

- Application software: Application software consists of standalone programs that solve a specific business need. E.g. Point of sole transaction processing, real time manufacturing control etc.

- Engineering/scientific software: Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

- Embedded software: Embedded software resides within a product or system and is used to implement and control features and functions for the end-user and for the system itself. Embedded software can perform limited and esoteric functions. e.g. digital functions in an automobile such as fuel control, dashboard displays, braking systems, etc.

- Product-line software: Designed to provide a specific capability for use by many different customers product-line software can focus on a limited and esoteric marketplace or address mass consumer markets. e.g. inventory control, word processing, spreadsheets, computer graphics, multimedia etc.

- WebApps (Web applications): "WebApps," span a wide array of applications. In their simplest form, webApps can be little more than a set of linked hypertext files that present information using text and limited graphics.

- AI software: Al software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. e.g. robotics.

■ **Software—New Categories**

- Ubiquitous computing: wireless networks

- Net sourcing: the Web as a computing engine

- Open source: "free" source code open to the computing community (a blessing, but also a potential curse!)

- Also …
    - Data mining
    - Grid computing
    - Cognitive machines
    - Software for nanotechnologies

1.4 SOFTWARE MYTHS

- Unlike ancient myths, software myths propagate misinformation and confusion that have caused serious problems for managers, technical people and customers.

Management Myths

- **Myth:** We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?
- **Reality:** The book of standards may very well exist, but is it used? Is it complete? In many cases, the answer is no.
- **Myth:** If we get behind schedule, we can add more programmers and catch up.
- **Reality:** Software development is not a mechanistic process like manufacturing.
- **Myth:** If we decide to outsource the software project to a third party, I can just relax and let that firm build it.
- **Reality:** If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it out sources software projects.

Customer Myths

- **Myth**: Project requirements continually change, but change can be easily accommodated because software is flexible.
- **Reality:** It is true that software requirements do change, but the impact of change varies with the time at which it is introduced.
- **Myth:** A general statement of objectives is sufficient to begin writing programs we can fill in the details later.
- **Reality:** Although a comprehensive and stable statement of requirements is not always possible, an ambiguous statement of objectives is a recipe for disaster.

Practitioner's Myths

- **Myth:** Once we write the program and get it to work our job is done.

- **Reality:** "The sooner you begin writing code, the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

- **Myth:** Until I get the program running, I have no way of assessing its quality.

- **Reality:** One of the most effective software quality assurance mechanisms can be applied from the inception of a project-the formal technical review.

- **Myth:** The only deliverable work product for a successful project is the working program.

- **Reality:** A working program is only one part of a software configuration that includes many elements.

- **Myth:** Software engineering will make us creates voluminous and unnecessary documentation and will invariably slow us down.

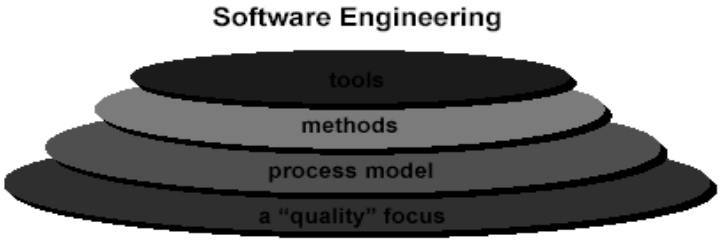
- **Reality:** Software engineering is not about creating documents. It is about creating quality. Better quality leads to reduced rework.

2. A GENERIC VIEW OF PROCESS

- **Chapter Overview**

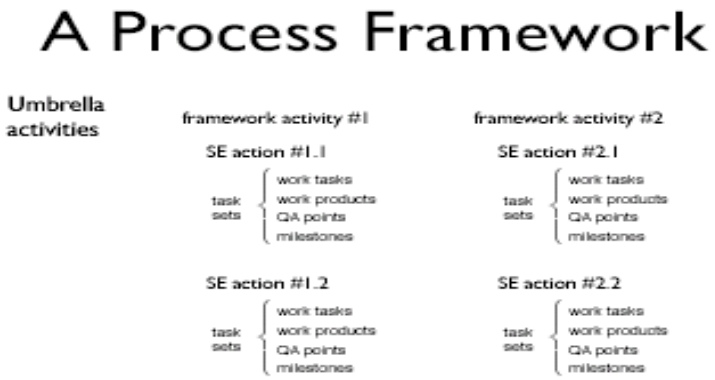
  - What is it? A software process-a series of predictable steps that leads to a timely, high-quality product.
  - Who does it? Managers, software engineers, and customers.
  - Why is it important? Provides stability, control, and organization to an otherwise chaotic activity.
  - What are the Steps? A handful of activities are common to all software process, details vary.
  - What is the work product? Programs, documents, and data.
  - Correct process? Assessment, quality deliverable.

2.1 SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY



- Focus on quality: must rest on an organizational commitment to quality.
- Process model: foundation for SE is the process layers, a frame work, models documents, data, reports etc.
- Methods: how to's, communication, requirement analysis design modeling program connection, testing and support.
- Tools: provide automated or semi automated support for the process and methods.

2.2 A PROCESS FRAMEWORK



■ **Process framework**

- Framework activities
  - Work tasks
  - Work products
  - Milestones & deliverables
  - QA checkpoints
  - Umbrella Activities

Framework Activities

- Communication: This framework activity involves heavy communication and collaboration with the customer and encompasses requirements gathering and other related activities.

- Planning: It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

- Modeling: This activity encompasses
  - Analysis of requirements
  - Design

- Construction: This activity combines
  - Code generation
  - Testing

- Deployment: The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

Umbrella Activities

- Software project tracking and control: allows the software team to assess progress against the project plan and take necessary action to maintain schedule.

- Formal technical reviews: Assesses software reengineering work products in an effort to uncover and remove errors before they are propagated to the next action or activity.

- Software quality assurance: Defines and conducts the activities required to ensure software quality.

- Software configuration management: manages the effects of change throughout the software process.

- Work product preparation and production: encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

- Reusability management: defines criteria for work product reuse and establishes mechanisms to achieve reusable components.

- Measurement: defines and collects process, project, and product measures that assist the team in delivering software that meets customers' needs; can be used in conjunction with all other framework and umbrella activities.

- Risk management: assesses risks that may affect the outcome of the project or the quality of the product.

■ **The Process Model: Adaptability**

- the framework activities will always be applied on every project ... BUT

- the tasks (and degree of rigor) for each activity will vary based on:
    - the type of project
    - characteristics of the project
    - common sense judgment; concurrence of the project team

2.3 THE CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

- The CMMI defines each process area in terms of "specific goals" and the "specific practices" required to achieve these goals.

    - *Specific goals* establish the characteristics that must exist if the activities implied by a process area are to be effective.

- *Specific practices* refine a goal into a set of process-related activities.

- The CMMI represents a process meta-model in tow different ways: (1) as continuous model and (2) as a staged model.

- **Capability levels:**

  - **Level 0: Incomplete.** The process area is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capacity**.**

  - **Level 1: Performed.** All of the specific goals of the process area have been satisfied. Work tasks required to produce defined work products are being conducted.

  - **Level 2: Managed.** All level 1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy; all people doing the work have access to adequate resources to get the job done; stakeholders re actively involved in the process area as require; all work tasks and work products are "monitored, controlled, and reviewed; and are evaluated for adherence to the process description"

  - **Level 3: Defined.** All level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines, and contributes work products, measures, and other process-improvement information to the organizational process assets"

  - **Level 4: Quantitatively managed.** All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as criteria in managing the process"

  - **Level 5: Optimized.** All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration".

2.4 PROCESS PATTERNS

- Process patterns define a set of activities, actions, work tasks, work products and/or related behaviors.

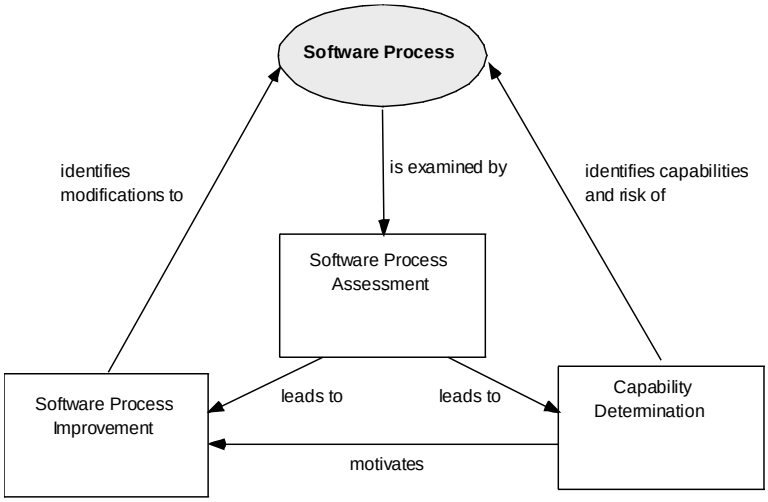
- A template is used to define a pattern.

- Typical examples:
  - Customer communication (a process activity)
  - Analysis (an action)

- Requirements gathering (a process task)
- Reviewing a work product (a process task)
- Design model (a work product)

2.5 PROCESS ASSESSMENT

- The process should be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering**.**



- Many different assessment options are available:

    - SCAMPI: provides a five-step process assessment model that incorporates initiating, diagnosing, establishing, acting, and learning.

    - CBA IPI: provides diagnostic technique for assessing the relative maturity of a software organization.

    - SPICE: standard defines a set of requirements for software process assessment.

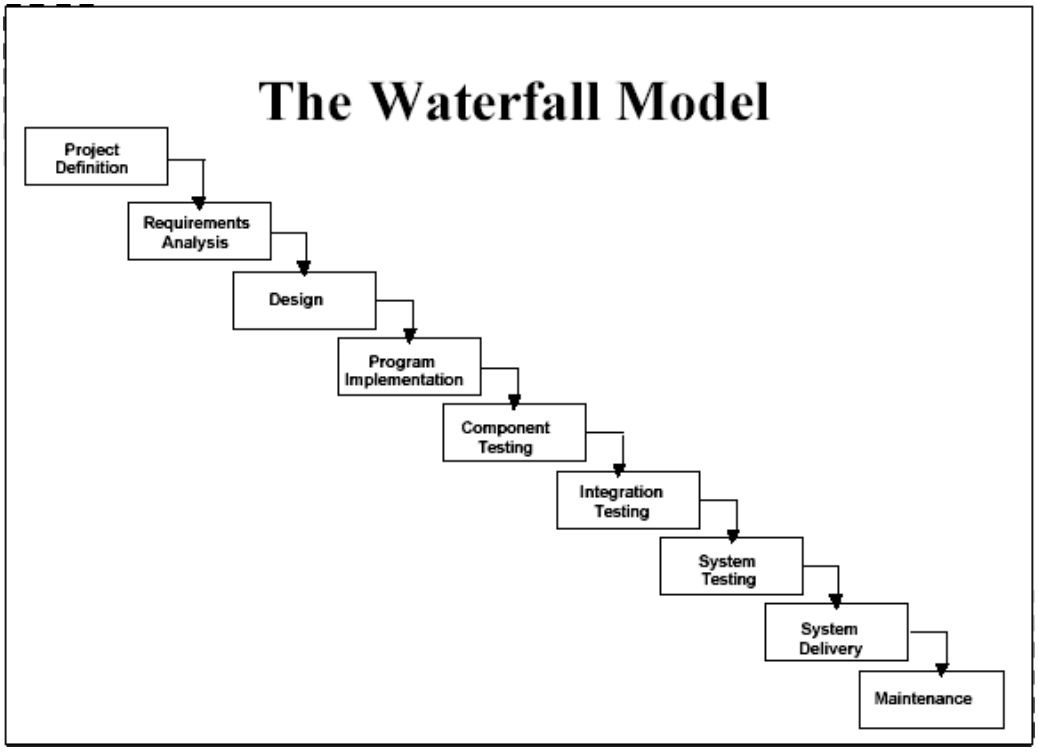
    - ISO 9001:2000: for software is a generic standard that applies to any organization that want improve the overall quality of the product systems, or services that it provides.

3. PROCESS MODELS

■ Prescriptive Models

- Prescriptive process models advocate an orderly approach to software engineering.
- *That leads to a few questions …*
    - If prescriptive process models strive for structure and order, are they inappropriate for a software world that thrives on change?
    - Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

3.1 The Waterfall Model



- The Waterfall model sometimes called the classic life cycle, suggests a systematic, sequential approach to software development.

- It is a oldest paradigm for software engineering.
- Most widely used though no longer state-of-art.
- Each step results in documentation.
- May be suited to for well-understood developments using familiar technology.
- Not suited to new, different systems because of specification uncertainty.
- Difficulty in accommodating change after the process has started.
- Can accommodate iteration but indirectly.
- Working version not available till late in process.
- Often get blocking states.

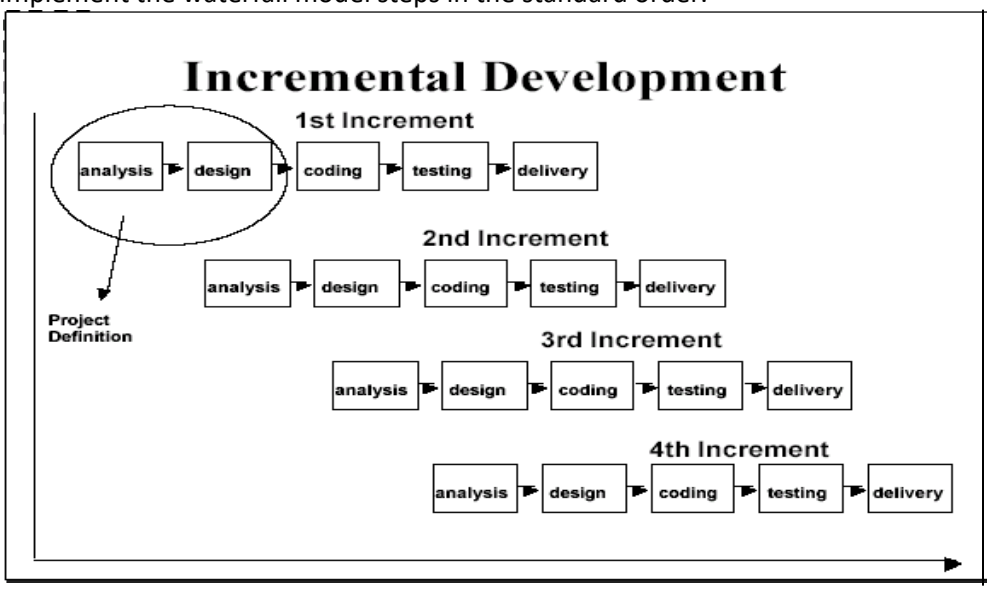
3.2 THE INCREMENTAL PROCESS MODELS

- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process.
- In addition, there may be a compelling need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.
- In such cases, a process model that is designed to produce the software in increments is chosen.

■ The Incremental Model

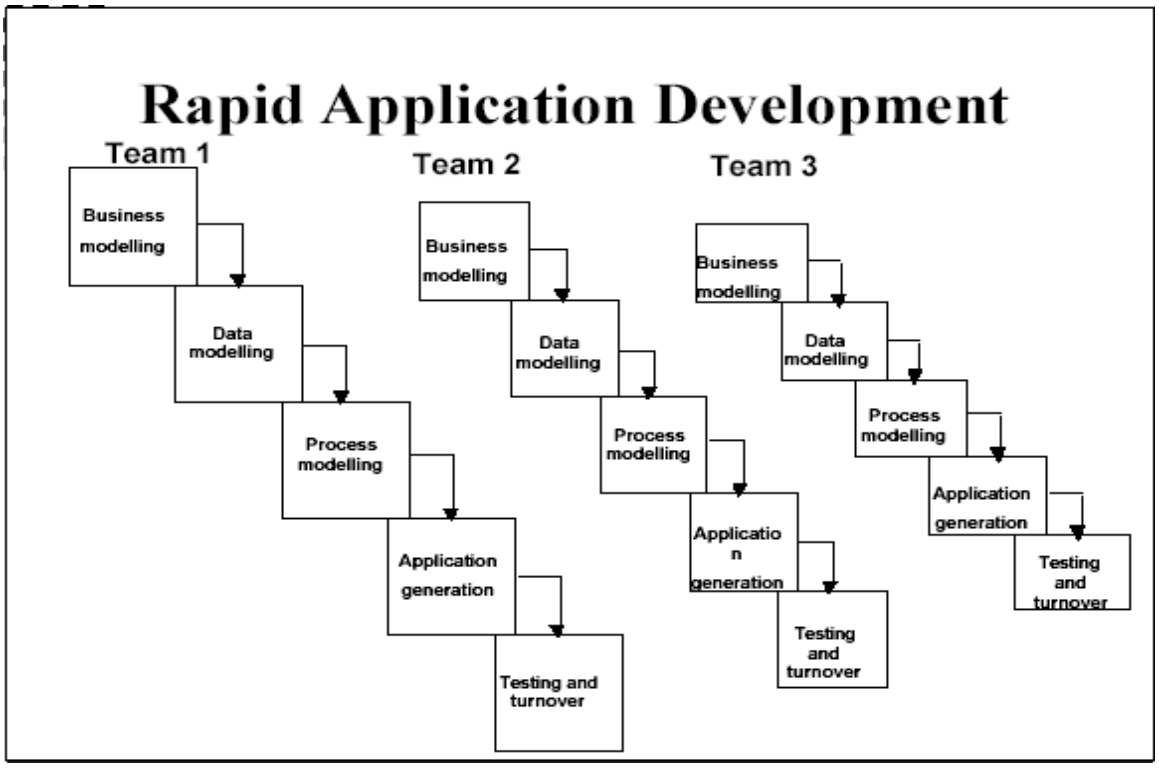
- Applies an iterative philosophy to the waterfall model.
- Divide functionality of system into increments and use a liner sequence of development on each increment.
- First increment delivered is usually the core product, i.e. only basic functionality.

- Reviews of each increment impact on design of later increments.
- Manages risk well.
- Extreme Programming (XP), and other Agile Methods, are incremental, but they do not implement the waterfall model steps in the standard order.



■ **The Rapid Application Development (RAD) Model**

- Similar to waterfall but uses a very short development cycle (60to90 days to completion).
- Uses component-based construction and emphasizes reuse and code generation.
- Use multiple teams on scaleable projects.
- Requires heavy resource.
- Requires developers and customers who are heavily committed.
- Performance can be a problem.
- Difficult to use with new technology



3.3 EVOLUTIONARY MODELS

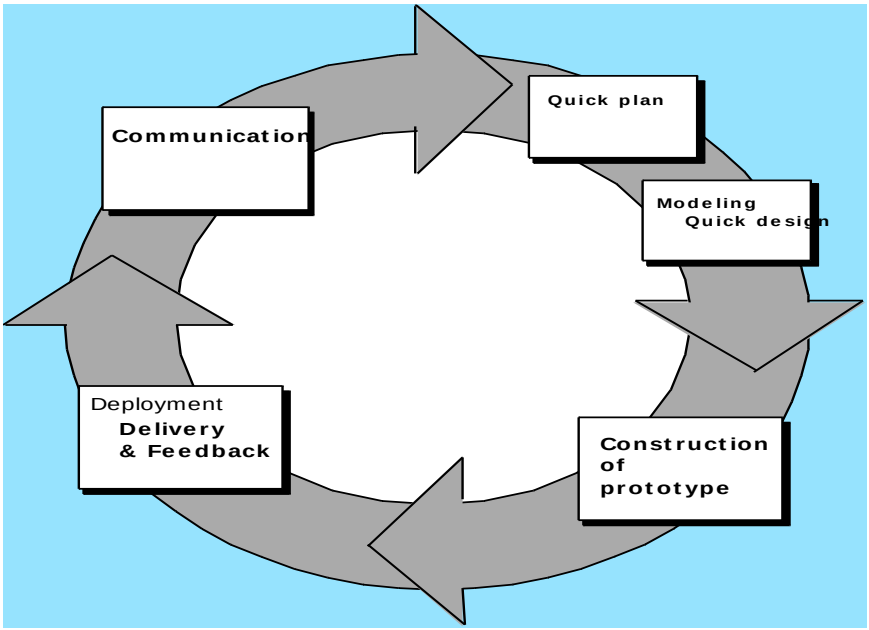
- Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

■ **Prototyping**

- Ideally mock-up serves as mechanism for identifying requirements.
- Users like the method, get a feeling for the actual system.
- Less ideally may be the basis for completed.
    - Prototypes often ignore quality/performance/maintenance issues.
    - May create pressure from users on deliver earlier.

- May use a less-than-ideal platform to deliver e.g Visual Basic – excellent for prototyping, may not be as effective in actual operation.

- Specifying requirements is often very difficult.
- Users don't know exactly what they want until they see it.
- Prototyping involves building a mock-up of the system and using to obtain for user feedback.
- Closely related to what are now called "Agile Methods"

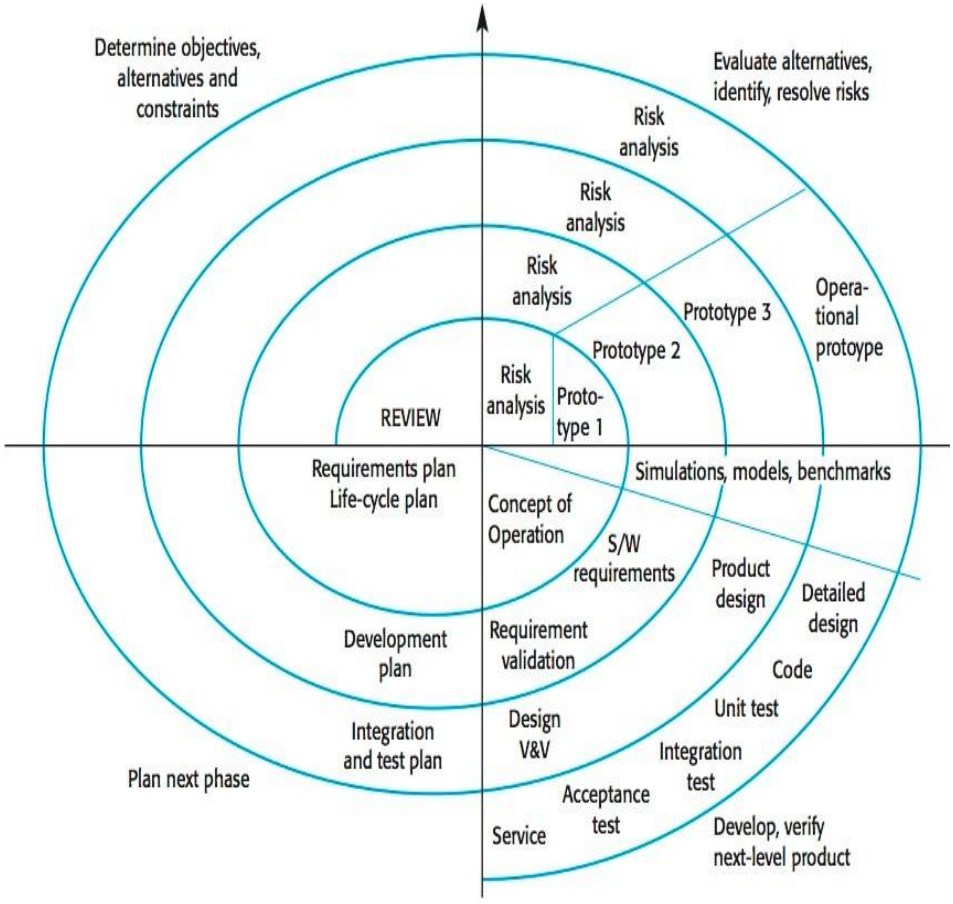


■ **The Spiral Model**

- Development cycles through multiple(3-6) task regions (6stage version).
  - Customer communication
  - Planning
  - Risk analysis
  - Engineering
  - Construction and release
  - Customer evaluation
- Incremental releases

- Early releases may be paper or prototypes.
- Later releases become more complicated

- Models software until it is no longer used
- Not a silver bullet, but considered to be one of the best approaches.
- Is a realistic approach to the problems of large scales software development?
- Can use prototyping during any phase in the evolution of product?
- Requires excellent management and risk assessment skills



■ Concurrent Development Model

- The concurrent development model, sometimes called concurrent engineering, can be represented schematically as a series of framework activities, software engineering actions and tasks, and their associated states.

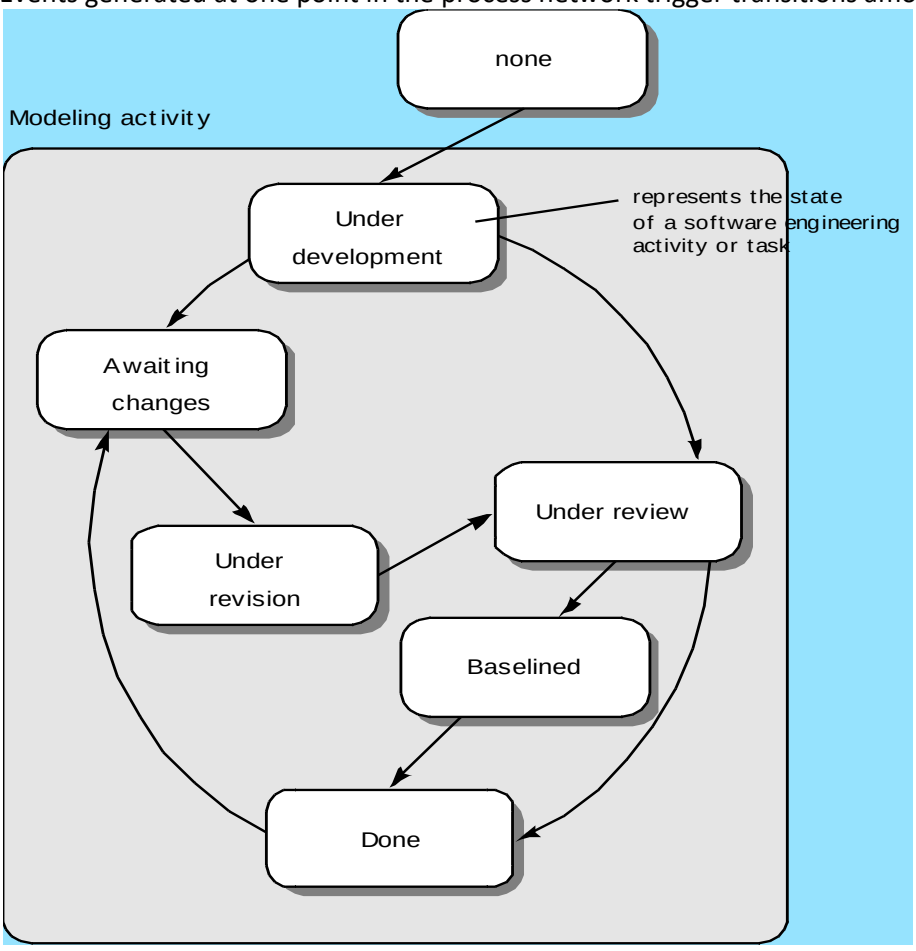
- The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities, action, or tasks.

- The concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project.

- Rather than confining software engineering activities, actions, and tasks to a sequence of events, it defines a network of activities.

- Each activity, action, or task on the network exists simultaneously with other activities, actions, or tasks.

- Events generated at one point in the process network trigger transitions among the states.



3.4 SPECIALIZED PROCESS MODELS

- *Component based development*—the process to apply when reuse is a development objective.

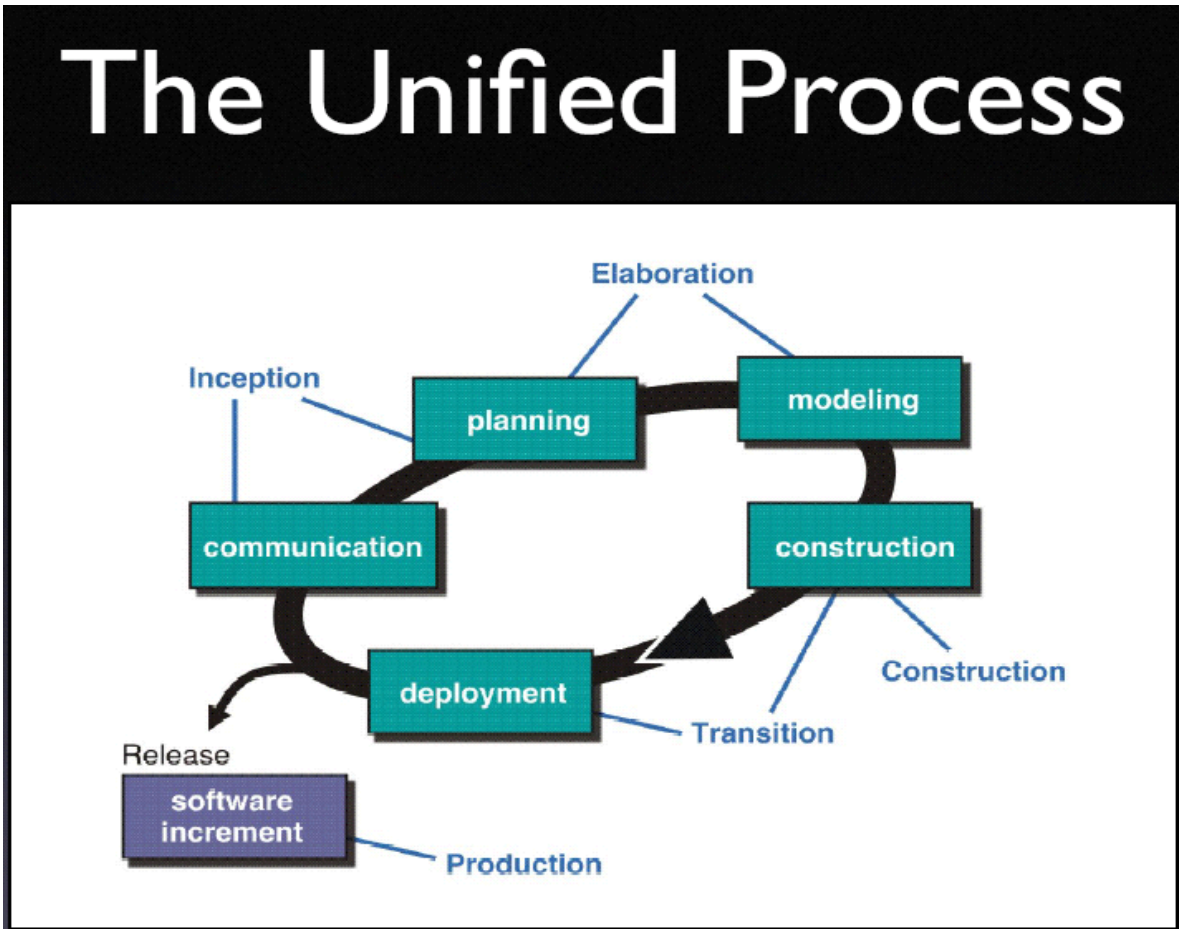
- *Formal methods*—emphasizes the mathematical specification of requirements.

- AOSD—provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*

3.5 THE UNIFIED PROCESS (UP)

- **Unified Process**

- a "use-case driven, architecture-centric, iterative and incremental" software process closely aligned with the Unified Modeling Language (UML)



■ **Unified Process Phases**

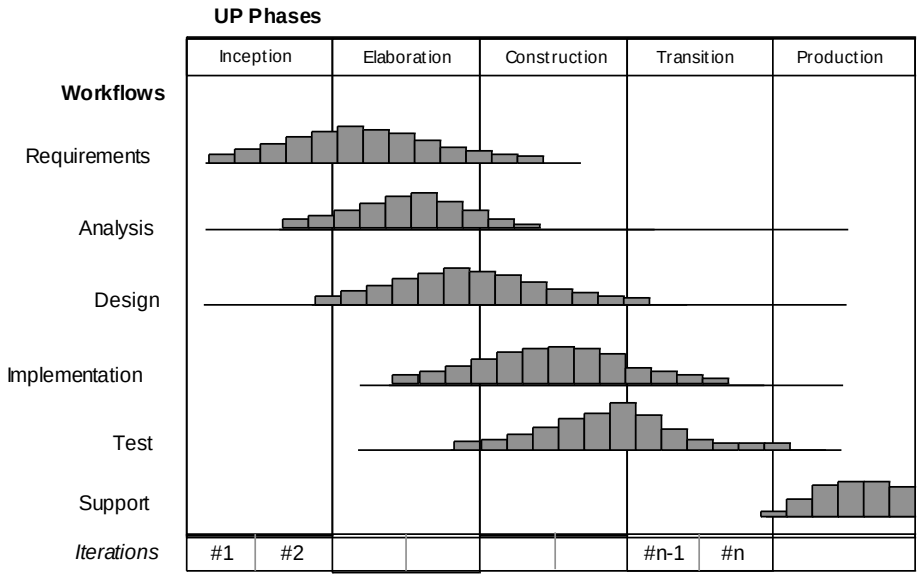
- The inception phases of the up encompass both customer communication and planning activities and emphasize the development and refinement of use-cases as a primary model.

- An elaboration phase that encompasses the customer's communication and modeling activities focusing on the creation of analysis and design models with an emphasis on class definitions and architectural representations.

- A construction phase that refines and translates the design model into implemented software components.

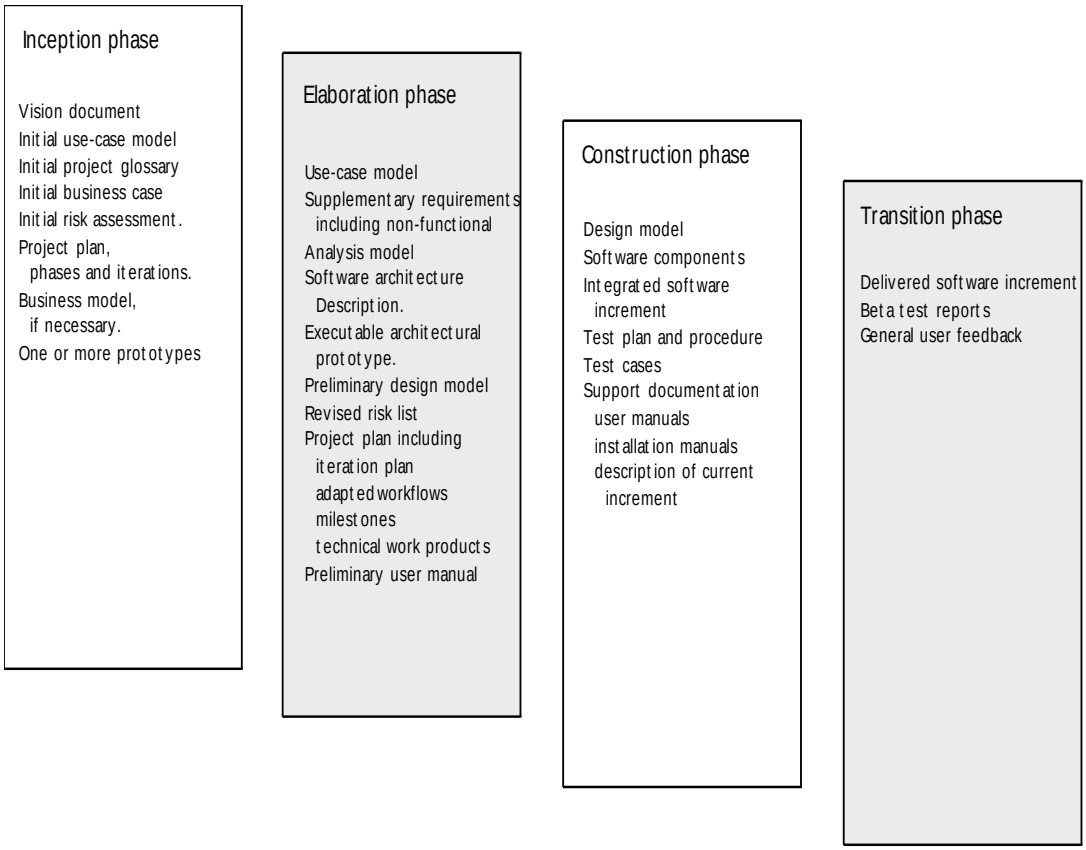
- A transition phase that transfers the software from the developer to the end-user for beta testing and acceptance.

- A production phase in which on-going monitoring and support are conducted.



■ UP Work Products

- The following figure illustrates the key work products produced as consequences of the four technical up phases.



4. SOFTWARE REQUIREMENTS

- **Contents:**
  - *Functional and non-functional requirements*
  - *User requirements*
  - *System requirements*
  - *Interface specification*
  - *The Software Requirements Document*

- **Requirements engineering:**

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed

- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process

■ **What is a requirement?**

- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification
- This is inevitable as requirements may serve a dual function

    - *May be the basis for a **bid** for a contract - therefore must be open to interpretation*
    - *May be the basis for the contract itself - therefore must be defined in detail*
    - *Both these statements may be called requirements*

- **Requirements abstraction (Davis, 1993)**

"If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organisation's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the *requirements document* for the system."

■ **Types of requirement**

- User requirements
    - *Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers*

- System requirements
    - *A structured document setting out detailed - descriptions of the system services. Written as a contract between client and contractor*

- Software specification
    - *A detailed software description which can serve as a basis for a design or implementation. Written for developers.*

- **Definitions and specifications**

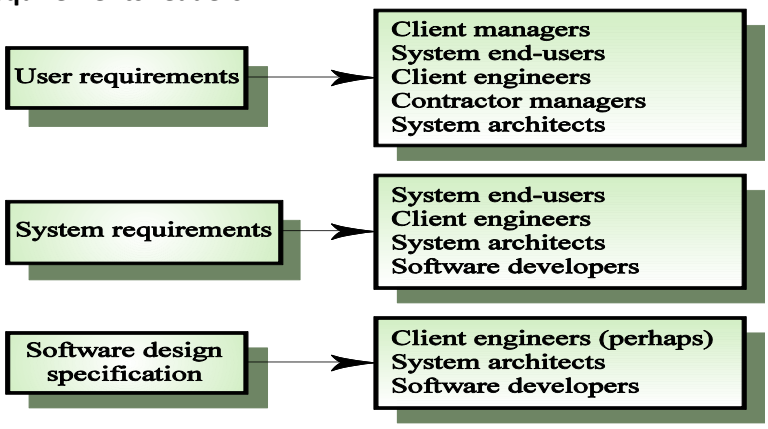
Requirements definition

1. The software must provide a means of representing and accessing external files created by other tools.

Requirements specification

1.1 The user should be provided with facilities to define the type of external files.
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specific icon on the userÕs display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When a user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

- **Requirements readers:**



- **Another classification of requirements**

  - Functional requirements
  - Non-functional requirements
  - Domain requirements

4.1 FUNCTIONAL REQUIREMENTS

- Describe functionality or system services, how the system should react to particular inputs and how the system should behave in particular situations.

- Depend on the type of software, expected users and the type of system where the software is used
- Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

- **Examples:**
  - *The user shall be able to search either all of the initial set of databases or select a subset from it.*
  - *The system shall provide appropriate viewers for the user to read documents in the document store.*
  - *Every order shall be allocated a unique identifier (ORDER_ID) which the user shall be able to copy to the account's permanent storage area.*
- Problems arise when requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term 'appropriate viewers' in the previous example.
  - *User intention* - special purpose viewer for each different document type
  - *Developer interpretation* - Provide a text viewer that shows the contents of the document

- Requirements should be complete and consistent
- Complete
  - They should include descriptions of **all** facilities required
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities

- In practice, it is impossible to produce a complete and consistent requirements document

4.2 NON-FUNCTIONAL REQUIREMENTS

- Define system properties and <u>constraints</u> e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

- Can be constraints on the process too
  - Use a particular CASE system, programming language or development method

- System maybe unusable if non-functional requirements are not satisfied (Critical)

■Non-functional classifications

- <u>Product requirements</u>

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

- Organisational requirements

  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements

  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

■ **Non-functional requirements examples**

- Product requirement
  - *It shall be possible for all necessary communication between the APSE and the user to be expressed in the standard Ada character set*

- Organisational requirement
  - *The system development process and deliverable documents shall conform to the process and deliverables defined in XYZCo-SP-STAN-95*

- External requirement
  - *The system shall not disclose any personal information about customers apart from their name and reference number to the operators of the system*

■ **Goals and requirements**

- Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.
- Goal
  - *A general intention of the user such as ease of use*
- Verifiable non-functional requirement
  - *A statement using some measure that can be objectively tested*
- Goals are helpful to developers as they convey the intentions of the system users

■ **Examples**

- **A system goal**
  - *The system should be easy to use by experienced controllers and should be organised in such a way that user errors are minimised*.

- **A verifiable non-functional requirement**

- *Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.*

■ **Requirements measures**

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

■ **Maintainability ?**

■ **Non-functional requirement conflicts**

- Conflicts between different non-functional requirements are common in complex systems
- Example: Spacecraft system
  - maximum storage required should be 4MB (fit onto ROM)
  - system should be written in Ada language (suitable of critical real-time software development)
  - It might be impossible to write an Ada program with the required functionality in less than 4MB
  - >> Trade-off needed.

4.3 DOMAIN REQUIREMENTS

- Derived from the application domain and describe system characteristics and features that reflect the domain
- May be new functional requirements, constraints on existing requirements or define specific computations
- If domain requirements are not satisfied, the system may be unworkable

■ **Domain requirements (examples)**

- Library system
  - *There shall be a standard user interface to all databases which shall be based on the Z39.50 standard.*
  - *Because of copyright restrictions, some documents must be deleted immediately on arrival. Depending on the user's requirements, these documents will either be printed locally on the system server for manually forwarding to the user or routed to a network printer*.

- Train Protection system
  - *The deceleration of the train shall be computed as:*
    Dtrain = Dcontrol + Dgradient

  *Where Dgradient is 9.81ms2 * compensated gradient/alpha and where the values of 9.81ms2 /alpha are known for different types of train*

■ **Domain requirements problems**

- Understand ability
  - Requirements are expressed in the language of the application domain
  - This is often not understood by software engineers developing the system

- Implicitness
  - Domain specialists understand the area so well that they do not think of making the domain requirements explicit.

4.4 USER REQUIREMENTS

- Should describe functional and non-functional requirements so that they *are understandable by system users* who don't have detailed technical knowledge
- User requirements are defined using natural language, tables and diagrams.

■ **Problems with natural language**

- Lack of clarity
  - Precision is difficult without making the document difficult to read
- Requirements confusion

- Functional and non-functional requirements tend to be mixed-up
- Requirements amalgamation
- Several different requirements may be expressed together

■ **Example: editor grid requirement**

- ***Grid facilities*** *To assist in the positioning of entities on a diagram, the user may turn on a grid in either centimetres or inches, via an option on the control panel. Initially, the grid is off. The grid may be turned on and off at any time during an editing session and can be toggled between inches and centimetres at any time. A grid option will be provided on the reduce-to-fit view but the number of grid lines shown will be reduced to avoid filling the smaller diagram with grid lines.*

■ **Requirement problems**

- Grid requirement mixes three different kinds of requirement
  - Conceptual functional requirement (the need for a grid)
  - Non-functional requirement (grid units)
  - Non-functional UI requirement (grid switching)
- Sometimes requirements include both conceptual and detailed information
  - the detailed information should be specified in the system requirement specification

■ **Structured presentation**

Grid facilities: The editor shall provide a grid facility where a matrix of horizontal and vertical lines provides a background to the editor window**.** This grid shall be a passive grid where the alignment of entities is the user's responsibility.

Rationale**:** A grid helps the user to create a tidy diagram with well-spaced entities. Although an active grid, where entities 'snap-to' grid lines can be useful, the positioning is imprecise. The user is the best person to decide where entities should be positioned.

■ **Guidelines for writing requirements**

- Invent a standard format and use it for all requirements
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements
- Use text highlighting to identify key parts of the requirement
- Avoid the use of computer jargon

4.5 SYSTEM REQUIREMENTS

- More detailed specifications of user requirements
- Serve as a basis for designing the system
- May be used as part of the system contract

- System requirements may be expressed using system models

■ Requirements and design

- In principle, requirements should state **WHAT** the system should do (and the design should describe how it does this)
- In practice, requirements and design are inseparable
    - A system architecture may be designed to structure the requirements
    - The system may inter-operate with other systems that generate design requirements
    - The use of a specific design may be a domain requirement

■ Natural Language specification: Problems

- Ambiguity
    - The readers and writers of the requirement must interpret the same words in the same way. NL is naturally ambiguous so this is very difficult
    - E.g. signs on an escalator:
        - 'Shoes must be worn'
        - 'Dogs must be carried'
- Over-flexibility
    - The same thing may be said in a number of different ways in the specification
- Lack of modularisation
    - NL structures are inadequate to structure system requirements

■ Alternatives to NL specification

Notation	Description
Structured natural language	This approach depends on defining standard forms or templates to express the requirements specification.
Design description languages	This approach uses a language like a programming language but with more abstract features to specify the requirements by defining an operational model of the system.
Graphical notations	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. Use-case descriptions (Jacobsen, Christerson et al., 1993) are one technique.
Mathematical specifications	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

■ Structured language specifications

- A limited form of natural language may be used to express requirements
- This removes some of the problems resulting from ambiguity and flexibility and imposes a degree of uniformity on a specification
- Often supported using a forms-based approach

■ Form-based specifications

- Definition of the function or entity
- Description of inputs and where they come from
- Description of outputs and where they go to
- Indication of other entities required
- Pre and post conditions (if appropriate)
- The side effects (if any)

■ PDL-based requirements definition

- Requirements may be defined operationally using a programming language like notation but with more flexibility of expression
- Most appropriate in two situations
    - Where an operation is specified as a sequence of actions and the order is important (when nested conditions and loops are involved)
    - When hardware and software interfaces have to be specified. Allows interface objects and types to be specified

■ PDL disadvantages

- PDL may not be sufficiently expressive to express the system functionality in an understandable way
- Notation is only understandable to people with programming language knowledge
- The requirement may be taken as a design specification rather than a model to help understand the system

4.6 Interface specification

- Most systems must operate with other systems and the operating interfaces must be specified as part of the requirements
- Three types of interface may have to be defined
    - Procedural interfaces
    - Data structures that are exchanged
    - Data representations
- Formal notations are an effective technique for interface specification

■ **PDL interface description**

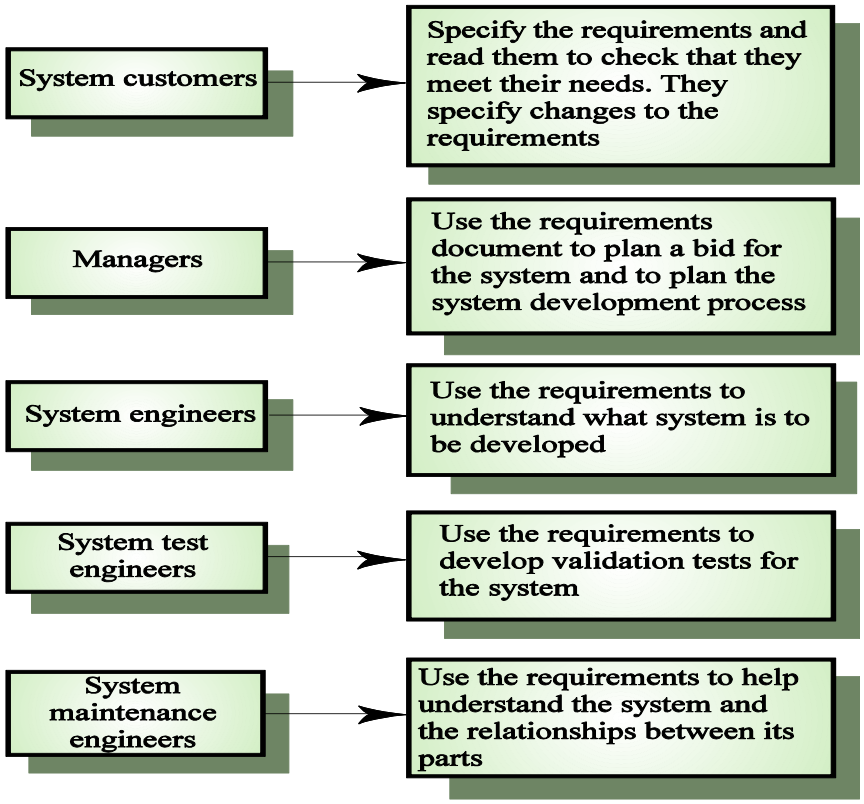
```
interface PrintServer {


// defines an abstract printer server

// requires:       interface Printer, interface PrintDoc

// provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter


        void initialize ( Printer p ) ;

        void print ( Printer p, PrintDoc d ) ;

        void displayPrintQueue ( Printer p ) ;

        void cancelPrintJob (Printer p, PrintDoc d) ;

        void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;

} //PrintServer
```

4.7 THE REQUIREMENTS DOCUMENT

- The requirements document is the official statement of what is required of the system developers

- Should include both a definition and a specification of requirements
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it

■ **Users of a requirements document**



■ **Requirements document requirements!**

- Specify external system behaviour
- Specify implementation constraints
- Easy to change
- Serve as reference tool for maintenance
- Record forethought about the life cycle of the system i.e. predict changes
- Characterise responses to unexpected events

■ **IEEE requirements standard**

- Introduction
- General description
- Specific requirements

- Appendices
- Index
- This is a generic structure that must be instantiated for specific systems

■ **REQUIREMENTS DOCUMENT STRUCTURE**

- Introduction
- Glossary
- User requirements definition
- System architecture
- System requirements specification
- System models
- System evolution
- Appendices
- Index

_____

5. REQUIREMENTS ENGINEERING PROCESSES

■ **Objectives**

The objective of this chapter is to discuss the activities involved in the requirements engineering process. When you study this chapter, you will:

- Understand the principal requirements of engineering activities and their relationships;
- Have been introduced to several techniques of requirements elicitation and analysis;
- understand the importance of requirements validation and how requirements reviews are used in this process;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

■ **Contents**

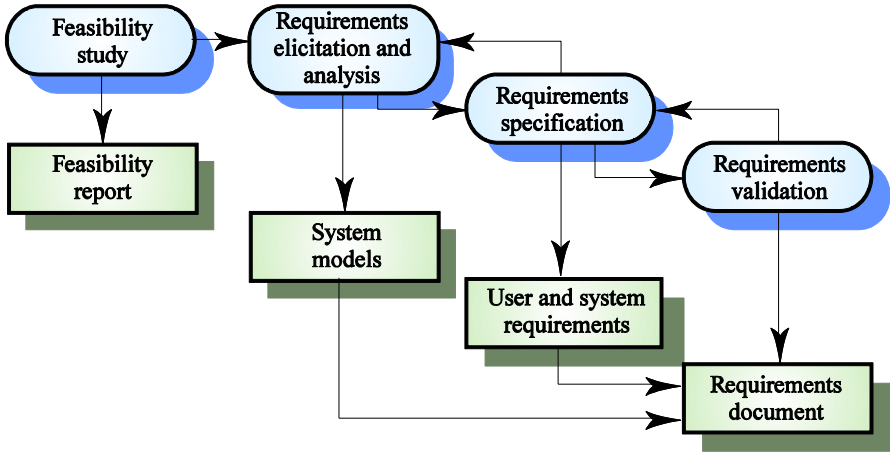
- Feasibility studies

- Requirements elicitation and analysis
- Requirements validation
- Requirements management

■ ***Requirements engineering processes***

- RE processes vary widely depending on
  - the application domain
  - the people involved and
  - the organisation developing the requirements

- However, there are a number of generic activities common to all processes
  - Feasibility study
  - Requirements elicitation and analysis
  - Requirements specification (documenting)
  - Requirements validation
  - Requirements management is an additional activity to manage changing requirements



5.1 FEASIBILITY STUDY

- A feasibility study decides whether or not the proposed system is worthwhile
- A short focused study that checks
  - If the system contributes to organisational objectives
  - If the system can be engineered using current technology and within budget
  - If the system can be integrated with other existing systems

■ ***Feasibility study implementation***

- Based on information assessment (what is required), information collection and report writing

- Questions for people in the organisation
    - What if the system wasn't implemented?
    - What are current process problems?
    - How will the proposed system help?
    - What will be the integration problems?
    - Is new technology needed? What skills?
    - What facilities must be supported by the proposed system?

■ ***Feasibility study report***

- The FS report should recommend whether or not the system development should continue.

- It may propose changes to the scope, budget, schedule and also suggest requirement changes

5.2 REQUIREMENTS ELICITATION AND ANALYSIS

- (requirements discovery)

- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints

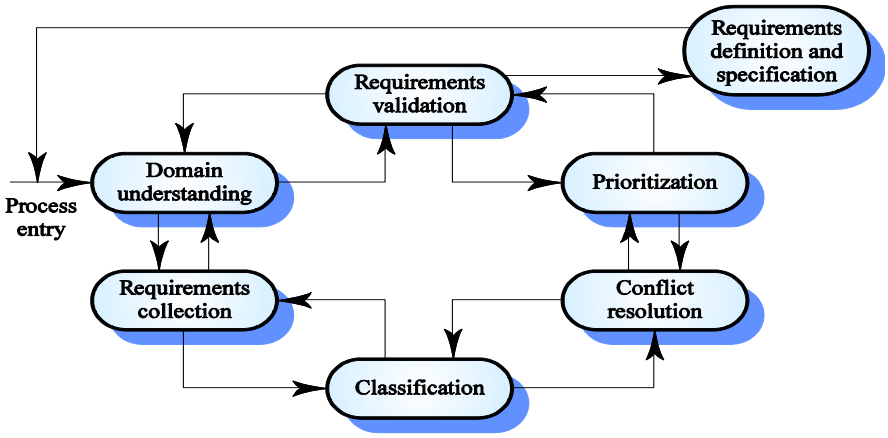
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*

■ ***Problems of requirements analysis***

- **Stakeholders** don't know what they really want
- Stakeholders express requirements in their own terms
- Different stakeholders may have conflicting requirements
- Organisational and political factors may influence the system requirements

- The requirements change during the analysis process. New stakeholders may emerge and the business environment change

■ ***The requirements analysis process***



■ ***The requirements analysis process activities***

- Domain understanding
- Requirements collection
- Classification
- Conflict resolution
- Prioritisation
- Requirements checking

■ ***Techniques for requirement elicitation and analysis***

- viewpoint-oriented elicitation
- ethnography
- scenarios
- structured analysis methods (system models)
- prototyping
- There is no universal method for requirement analysis!

■ ***System models***

- Different models may be produced during the requirements analysis activity
- Will be covered **later**

■ ***Viewpoint-oriented elicitation***

- Stakeholders represent different ways of looking at a problem or problem viewpoints
- This multi-perspective analysis is important as there is no single correct way to analyse system requirements

■ ***Scenarios***

- Descriptions of how a system is used in practice
- Helpful in requirements elicitation as people can relate to these more easily than an abstract statement of what they require from a system
- Useful for adding detail to an outline requirements description

■ ***Scenario descriptions***

- System state at the beginning of the scenario
- Normal flow of events in the scenario
- What can go wrong and how this is handled
- Other concurrent activities
- System state on completion of the scenario

■ ***Scenario based techniques***

- use cases!
- event scenarios
  - Event scenarios may be used to describe how a system responds to the occurrence of some particular event
  - Used in the Viewpoint Oriented Requirements Definition (VORD) method.

■ ***Ethnography***

- An analyst spends time observing and analysing how people actually work
- People do not have to explain their work
- Social and organisational factors of importance may be observed
- Identifies implicit system requirements
- *Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models*

■ ***Scope of ethnography***

- Requirements that are derived from the way that people actually work rather than the way in which process definitions suggest that they ought to work
  - e.g. air-traffic controllers switch off flight path conflict alarms

- Requirements that are derived from cooperation and awareness of other people's activities
    - e.g. predict no. of aircraft entering their sector by getting information from neighbouring controllers and plan accordingly

- Not suitable for using alone, has to be combined with some other technique

5.3 REQUIREMENTS VALIDATION

- Concerned with showing that the requirements define the system that the customer really wants
- Requirements error costs are high so validation is very important
    - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error

■ ***Types of requirements checking***

- *Validity*. Does the system provide the functions which best support the customer's needs?
- *Consistency*. Are there any requirements conflicts?
- *Completeness*. Are all functions required by the customer included?
- *Realism*. Can the requirements be implemented given available budget and technology
- *Verifiability*. Can the requirements be checked

■ ***Requirements validation techniques***

- Requirements reviews
    - Systematic manual analysis of the requirements
- Prototyping
    - Using an executable model of the system to check requirements.
- Test-case generation
    - Developing tests for requirements to check testability
- Automated consistency analysis
    - Checking the consistency of a structured requirements description

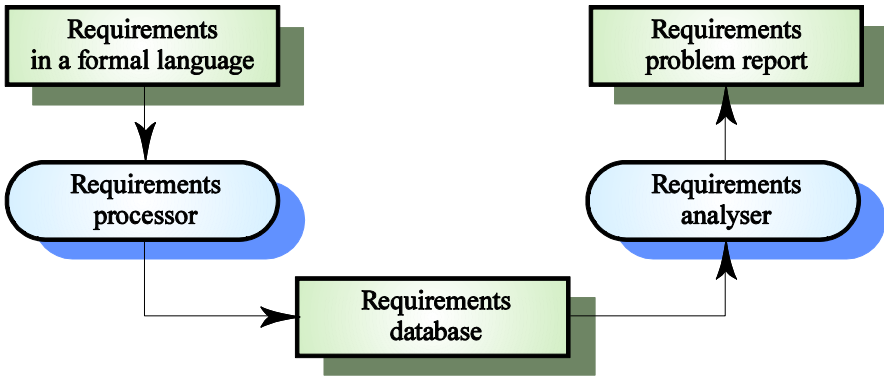
■ ***Requirements reviews***

- Regular reviews <u>while</u> the requirements definition is being formulated
- Both client and contractor staff should be involved in reviews
- Reviews may be **formal** (with completed documents) or **informal**. Good communications between developers, customers and users can resolve problems at an early stage

■ ***Review checks***

- Verifiability. Is the requirement realistically testable?
- Comprehensibility. Is the requirement properly understood?
- Traceability. Is the origin of the requirement clearly stated?
- Adaptability. Can the requirement be changed without a large impact on other requirements?

■ ***Automated consistency checking***



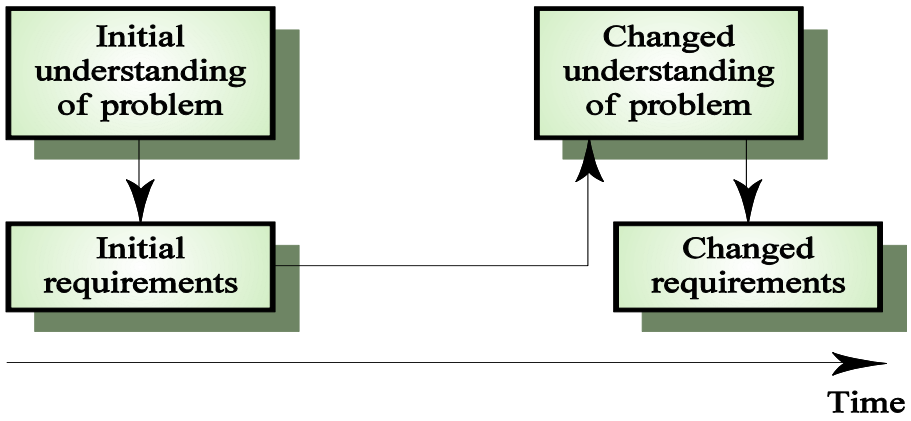
5.4 REQUIREMENTS MANAGEMENT

- Requirements management is the process of managing changing requirements during the requirements engineering process and system development
- Requirements are inevitably incomplete and inconsistent
    - New requirements emerge during the process as business needs change and a better understanding of the system is developed
    - Different viewpoints have different requirements and these are often contradictory

■ ***Why requirements change***

- The priority of requirements from different viewpoints changes during the development process
- System customers may specify requirements from a business perspective that conflict with end-user requirements
- The business and technical environment of the system changes during its development

■ ***Requirements evolution***



■ ***Enduring and volatile requirements***

- **Enduring requirements.** Stable requirements derived from the core activity of the customer organisation.
  - May be derived from domain models
  - E.g. a hospital will always have doctors, nurses, etc.

- **Volatile requirements.** Requirements which change during development or when the system is in use.
  - E.g. In a hospital, requirements derived from health-care policy

■ ***Classification of volatile requirements***

- Mutable requirements
  - Requirements that change due to the system's environment
- Emergent requirements
  - Requirements that emerge as understanding of the system develops
- Consequential requirements
  - Requirements that result from the introduction of the computer system
- Compatibility requirements
  - Requirements that depend on other systems or organisational processes

■ ***Requirements management planning***

- During the requirements engineering process, you have to plan:
  - Requirements identification
    - How requirements are individually identified
  - A change management process
    - The process followed when analysing a requirements change
  - Traceability policies
    - The amount of information about requirements relationships that is maintained
  - CASE tool support

- The tool support required to help manage requirements change

■ *Traceability*

- Traceability is concerned with the relationships between requirements, their sources and the system design
- Source traceability
  - Links from requirements to stakeholders who proposed these requirements
- Requirements traceability
  - Links between dependent requirements
- Design traceability
  - Links from the requirements to the design

▪ *A traceability matrix*

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			R		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

■ *CASE tool support*

- Requirements storage
  - Requirements should be managed in a secure, managed data store
  - We need requirement databases!
- Change management
  - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated
- Traceability management
  - Automated retrieval of the links between requirements

■ ***Requirements change management***

- Should apply to all proposed changes to the requirements
- Principal stages
    - Problem analysis. Discuss requirements problem and propose change
    - Change analysis and costing. Assess effects of change on other requirements
    - Change implementation. Modify requirements document and other documents to reflect change

---

6. SYSTEM MODELS

■ **Objectives**

The objective of this chapter is to introduce a number of system models that may be developed during the requirements engineering process. When you have study the chapter, you will:

- Understand why its is important to establish the boundaries of a system and model its context;
- Understand the concepts of behavioural modeling, data modeling and object modeling;
- Have been introduced to some of the notations defined in the Unified Modeling Language (UML) and how these notations may be used to develop system models.

■ **Contents**

- Context models
- Behavioural models

- Data models
- Object models
- Structured methods

■ ***System models***

- Abstract descriptions of systems whose requirements are being analysed
    - used to develop an understanding of the existing system or to specify the required system
    - used to communicate with others
    - they simplify the system (by leaving out details) and emphasise certain characteristics

■ ***Different perspectives***

- Different models present the system from different perspectives

    - External perspective - showing the system's context or environment
    - Behavioural perspective - showing the behaviour of the system
    - Structural perspective - showing the system or data architecture

■ ***Model types***

- *(Different approaches to abstraction)*

- **Data flow model** showing how the data is processed at different stages.

- **Composition model** showing how entities are composed of other entities.

- **Architectural model** showing principal sub-systems.

- **Classification model** showing how entities have common characteristics.

- **Stimulus/response model** showing the system's reaction to events.

6.1 CONTEXT MODELS

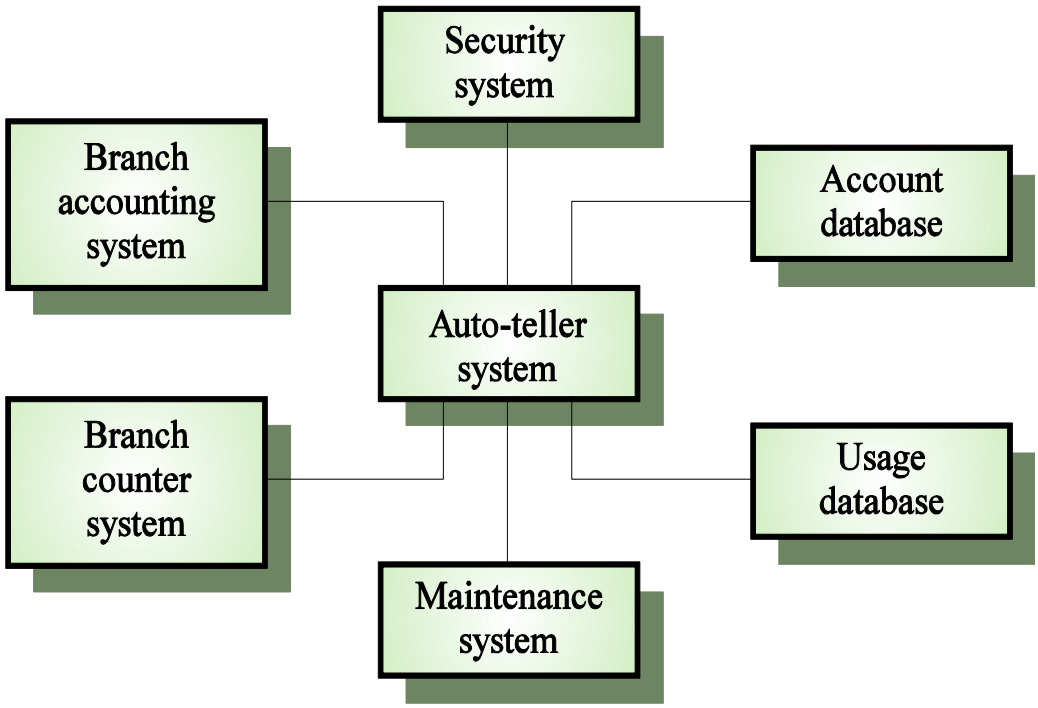
- Used to illustrate the operational context of a system - show what lies outside the system boundaries.

- Not only technical factors affect system boundary positioning

- Social and organisational concerns

- Architectural models show the system and its relationship with other systems.

■ ***Example: Context of an ATM system***



- Architectural models do not describe the system's relationships with the other systems in the environment

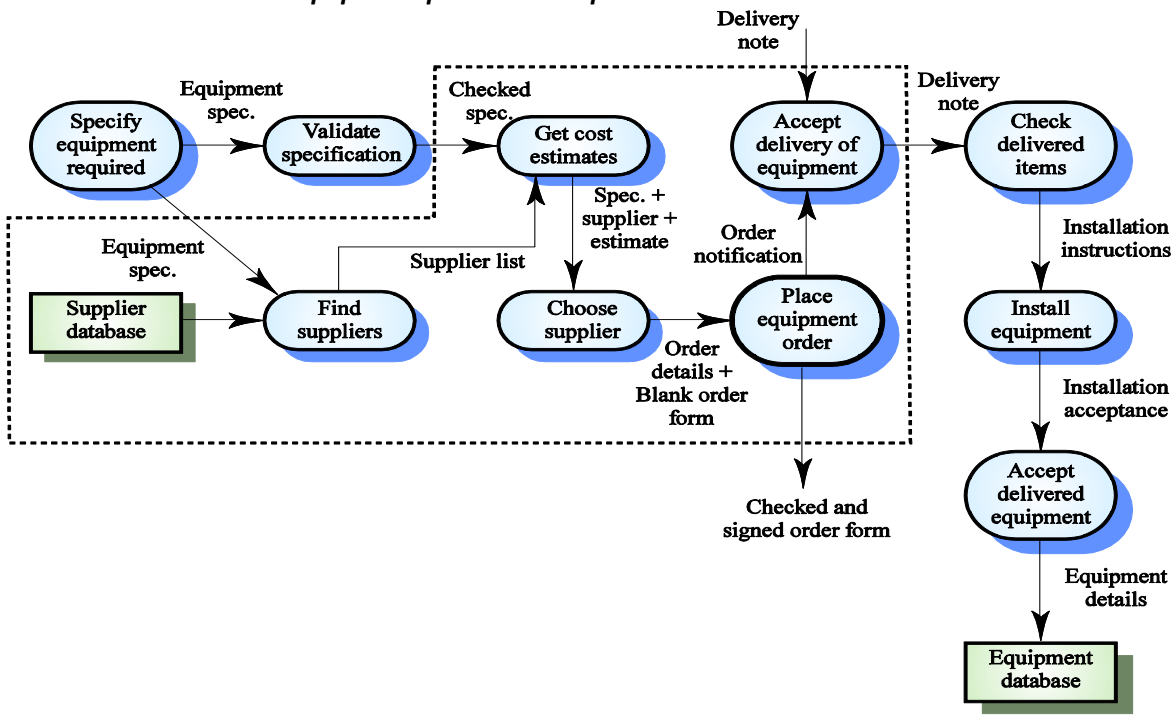
- Hence, architectural models are supplemented by other models such as
  - process models
  - data-flow models

- Process models

  - Process models show the overall process and the processes that are supported by the system

- Data flow models may be used to show the processes and the flow of information from one process to another

■ ***Equipment procurement process***



6.2 BEHAVIOURAL MODELS

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
  - Data processing models that show how data is processed as it moves through the system;
  - State machine models that show the systems response to events.

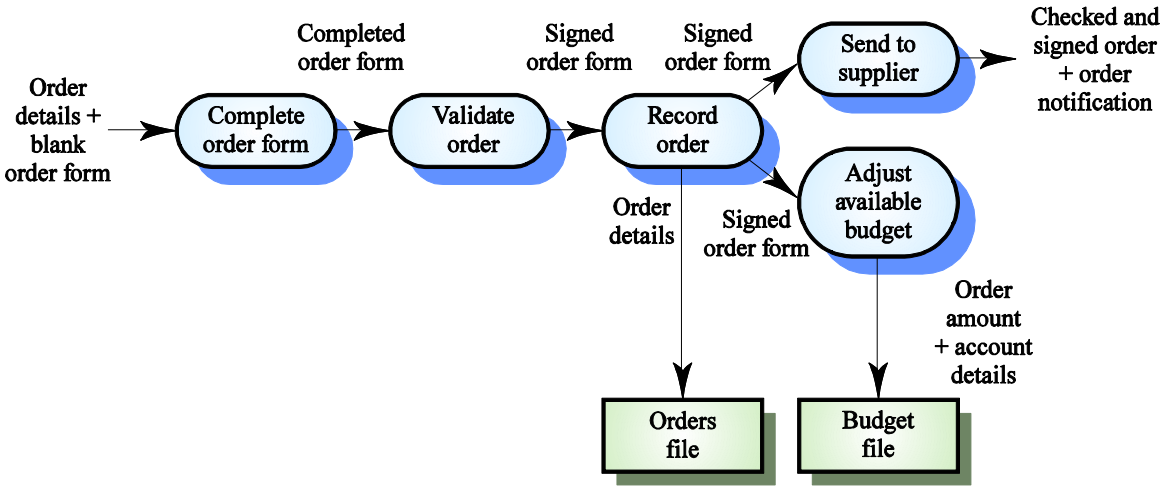
■ ***Data-processing models***

- Data flow diagrams (DFD) are used to model the system's data processing
- These show the processing steps as data flows through a system
- Intrinsic part of many analysis methods
- Simple and intuitive notation that customers can understand easily
- (different notations are used in different methods for drawing DFDs)

■ ***Elements of a DFD***

- Processes
  - Change the data. Each process has one or more inputs and outputs
- Data stores
  - used by processes to store and retrieve data (files, DBs)
- Data flows
  - movement of data among processes and data stores
- External entities
  - outside things which are sources or destinations of data to the system

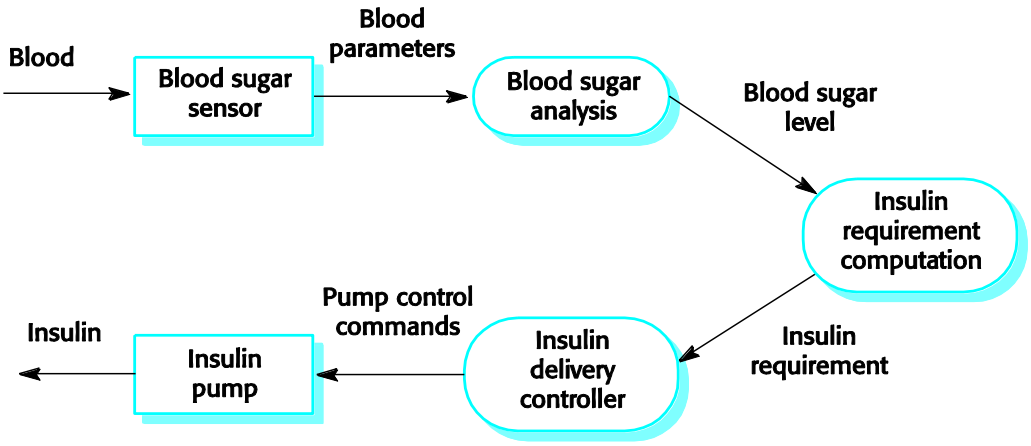
■ ***Example: Order processing DFD***



■ ***Data flow diagrams***

- DFDs model the system from a functional perspective.
- Tracking and documenting how the data associated with a process is helpful to develop an overall understanding of the system.
- Data flow diagrams may also be used in showing the data exchange between a system and other systems in its environment.

■ ***Insulin pump DFD***



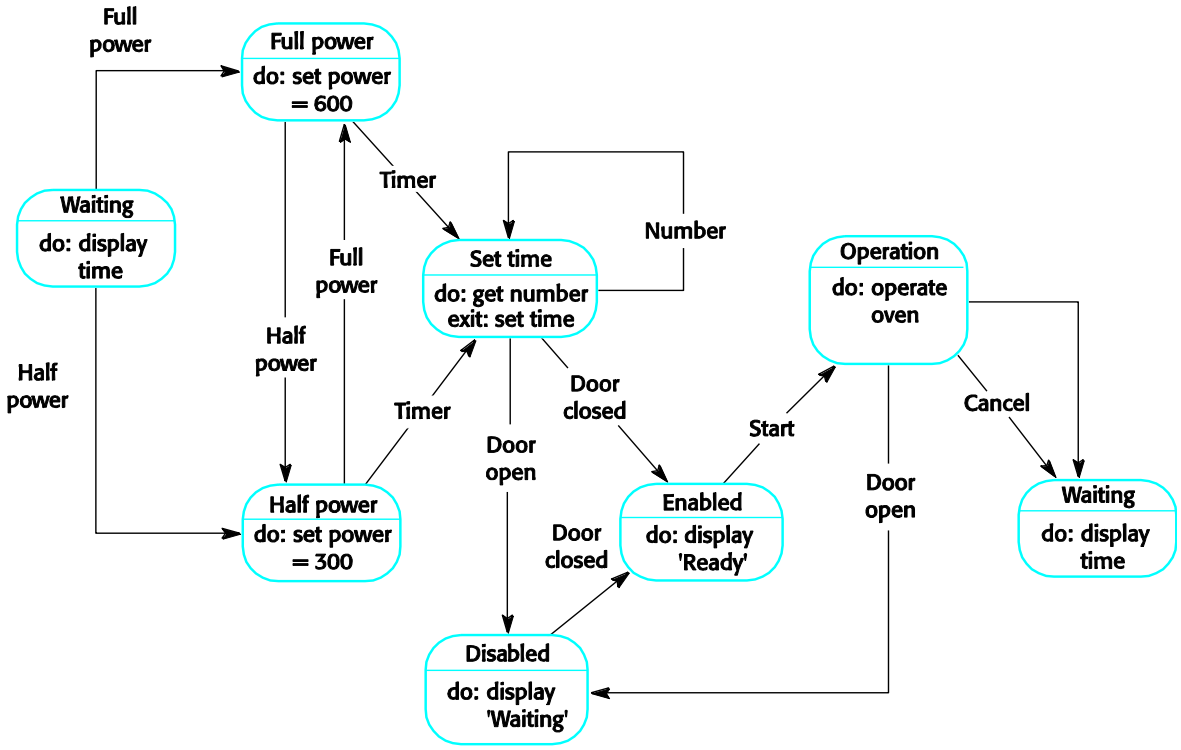
■ ***State machine models***

- These model the behaviour of the system in response to external and internal events.
- They show the system's responses to stimuli so are often used for modelling real-time systems.
- State charts are an integral part of the UML and are used to represent state machine models.

■ ***State charts***

- Allow the decomposition of a model into sub-models (see following slide).
- A brief description of the actions is included following the 'do' in each state.
- Can be complemented by tables describing the states and the stimuli.

■ ***Microwave oven model***



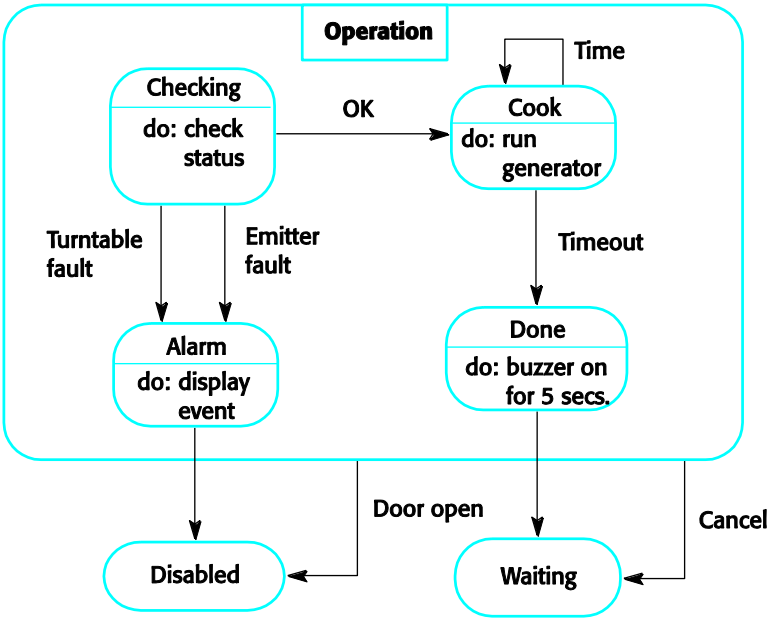
■ ***Microwave oven state description***

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows ‘Half power’.
Full power	The oven power is set to 600 watts. The display shows ‘Full power’.
Set time	The cooking time is set to the user’s input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows ‘Not ready’.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows ‘Ready to cook’.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows ‘Cooking complete’ while buzzer is sounding.

■ ***Microwave oven stimuli***

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

■ ***Microwave oven operation***



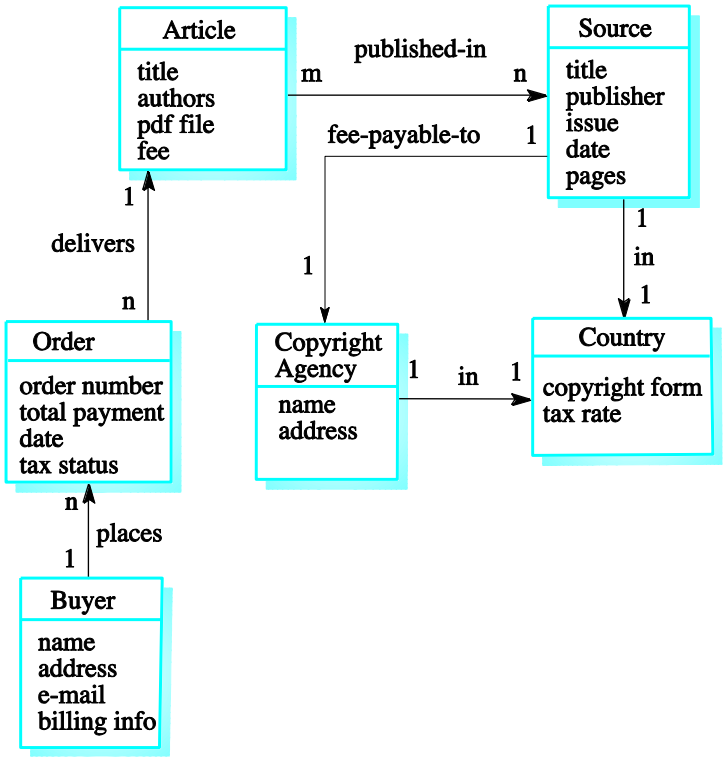
6.3 DATA MODELS

- Also called semantic data models

■ ***Semantic data models***

- Used to describe the logical structure of data processed by the system.
- An *entity-relation-attribute* model sets out the entities in the system, the relationships between these entities and the entity attributes
- Widely used in database design. Can readily be implemented using relational databases.

■ ***Library semantic model***



■ ***Data dictionaries***

- Data dictionaries are lists of all of the names used in the system models. Descriptions of the entities, relationships and attributes are also included.
- Advantages
  - Support name management and avoid duplication;
  - Store of organisational knowledge linking analysis, design and implementation;
- Many CASE workbenches support data dictionaries.

■ ***Data dictionary entries***

Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002

authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a copy of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

6.4 OBJECT MODELS

- Object models describe the system in terms of object classes and their associations.
- An object class is an abstraction over a set of objects with common attributes and the services (operations) provided by each object.
- Various object models may be produced
    - Inheritance models;
    - Aggregation models;
    - Interaction models.
- Natural ways of reflecting the real-world entities manipulated by the system
- More abstract entities are more difficult to model using this approach
- Object class identification is recognised as a difficult process requiring a deep understanding of the application domain
- Object classes reflecting domain entities are reusable across systems

■ Inheritance models

- Organise the domain object classes into a hierarchy.
- Classes at the top of the hierarchy reflect the common features of all classes.
- Object classes inherit their attributes and services from one or more super-classes. These may then be specialised as necessary.
- Class hierarchy design can be a difficult process if duplication in different branches is to be avoided.

■ Object models and the UML

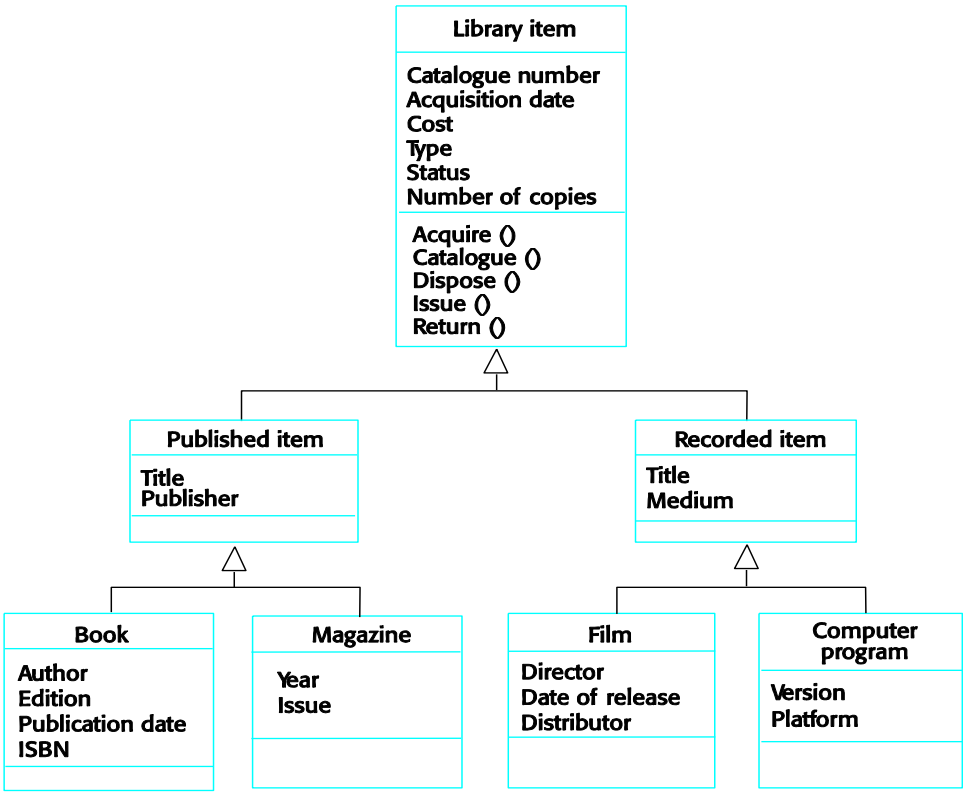
- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
    - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
    - Relationships between object classes (known as associations) are shown as lines linking objects;

- Inheritance is referred to as generalisation and is shown ‘upwards’ rather than ‘downwards’ in a hierarchy.

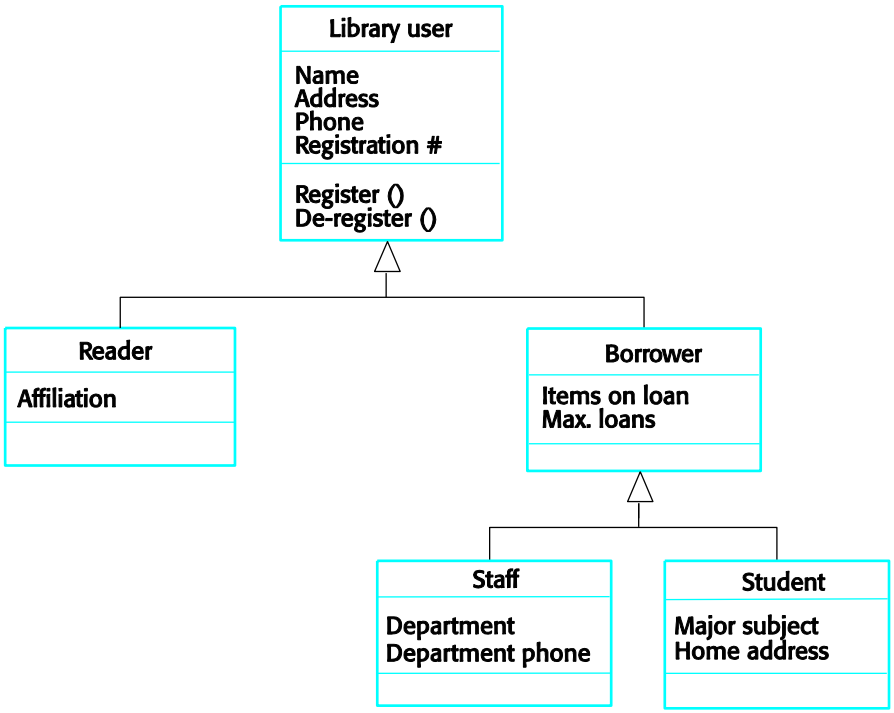
■ ***Object models and the UML***

- The UML is a standard representation devised by the developers of widely used object-oriented analysis and design methods.
- It has become an effective standard for object-oriented modelling.
- Notation
  - Object classes are rectangles with the name at the top, attributes in the middle section and operations in the bottom section;
  - Relationships between object classes (known as associations) are shown as lines linking objects;
  - Inheritance is referred to as generalisation and is shown ‘upwards’ rather than ‘downwards’ in a hierarchy.

■ ***Library class hierarchy***



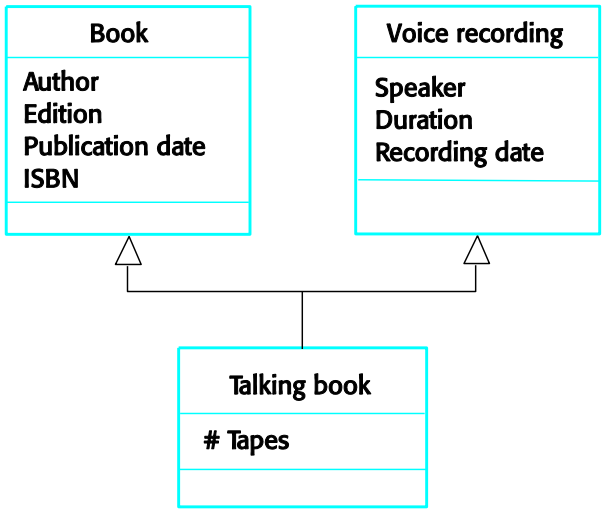
■ ***User class hierarchy***



- ***Multiple inheritance***

- Rather than inheriting the attributes and services from a single parent class, a system which supports multiple inheritance allows object classes to inherit from several super-classes.
- This can lead to semantic conflicts where attributes/services with the same name in different super-classes have different semantics.
- Multiple inheritance makes class hierarchy reorganisation more complex.

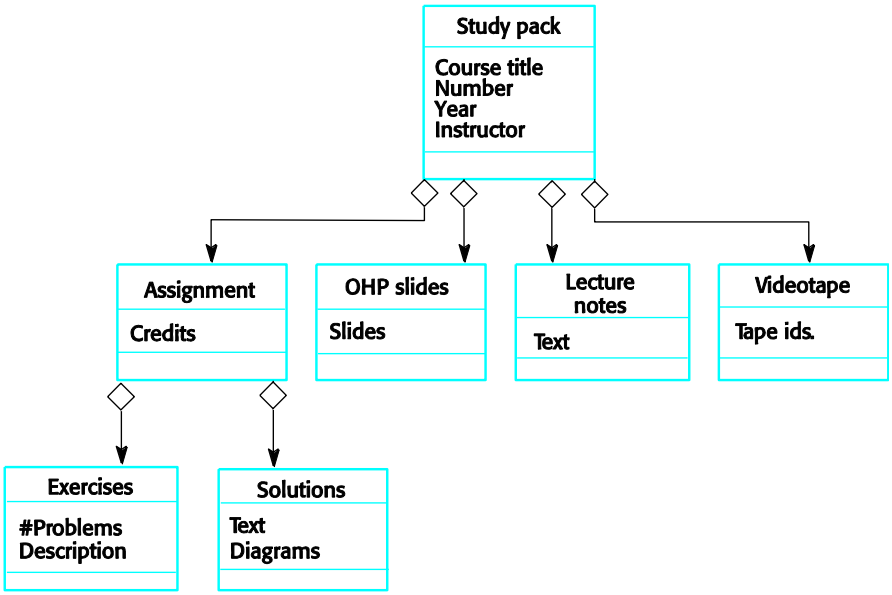
  - ***Multiple inheritance***



■ ***Object aggregation***

- An aggregation model shows how classes that are collections are composed of other classes.
- Aggregation models are similar to the part-of relationship in semantic data models.

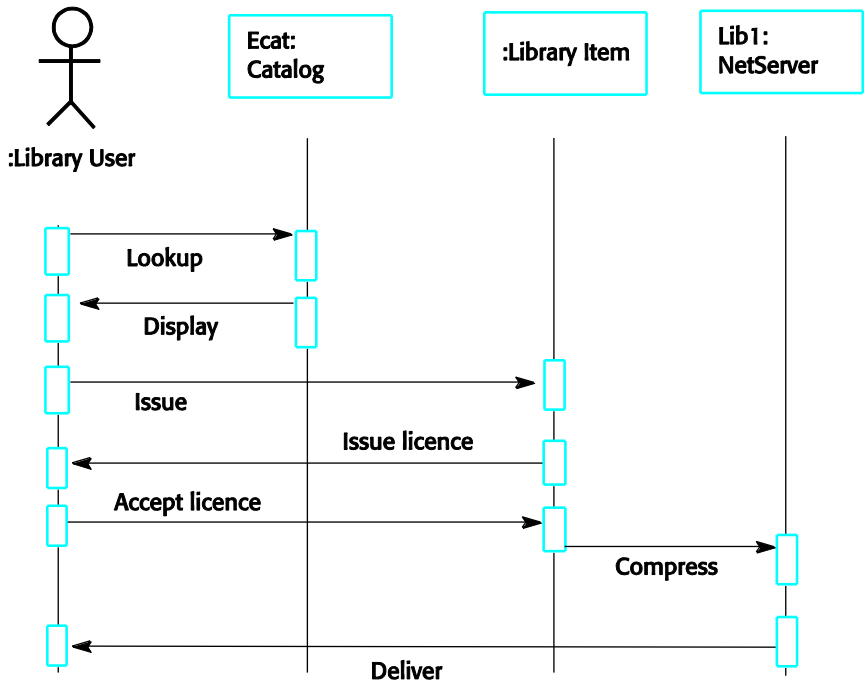
■ ***Object aggregation***



■ ***Object behaviour modelling***

- A behavioural model shows the interactions between objects to produce some particular system behaviour that is specified as a use-case.
- Sequence diagrams (or collaboration diagrams) in the UML are used to model interaction between objects.

■ ***Issue of electronic items***



6.5 STRUCTURED METHODS

- Structured methods incorporate system modelling as an inherent part of the method.
- Methods define:
  - a set of models
  - a process for deriving these models and
  - rules and guidelines that should apply to the models.
- CASE tools support system modelling as part of a structured method.

■ ***Method weaknesses***

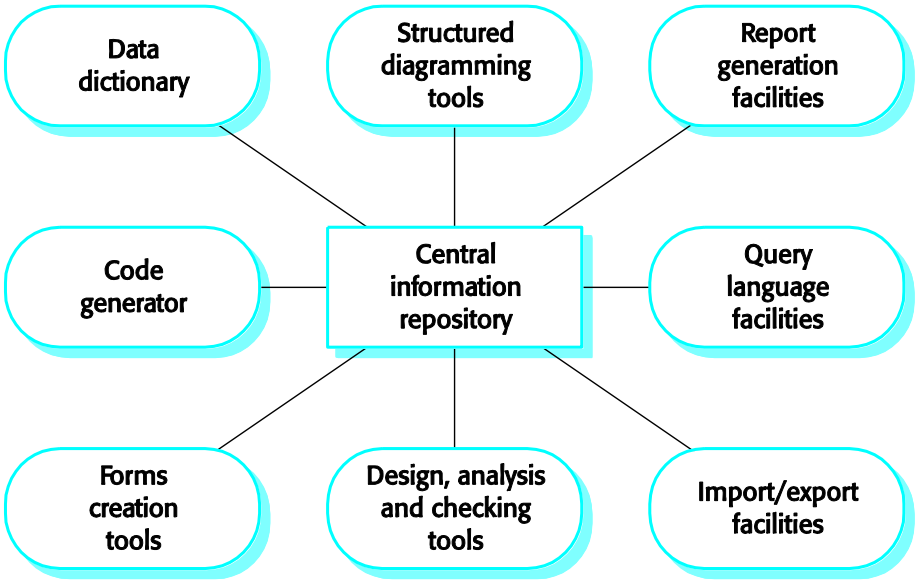
- Non-functional system requirements not modelled
- Appropriateness Do not usually include information about whether a method is appropriate for a given problem.
- Too much documentation
- Can be too detailed and difficult for users to understand.

■ ***CASE workbenches***

- A coherent set of tools that is designed to support related software process activities such as analysis, design or testing.
- Analysis and design workbenches support system modelling during both requirements engineering and system design.

- These workbenches may support a specific design method or may provide support for a creating several different types of system model.

■ ***An analysis and design workbench***



■ ***Analysis workbench components***

- Diagram editors
- Model analysis and checking tools
- Repository and associated query language
- Data dictionary
- Report definition and generation tools
- Forms definition tools
- Import/export translators
- Code generation tools

_____

7. DESIGN ENGINEERING

- The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence.

- Diversification and Convergence - the qualities which demand intuition and judgment are based on experience.

    - Principles and heuristics that guide the way the model is evolved.
    - Set of criteria that enables quality to be judge.
    - Process of iteration that ultimately leads to final design representation.

- Design engineering for computer software changes continually as new methods, better analysis, and broader understanding evolve. Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines.

7.1 DESIGN PROCESS AND DESIGN QUALITY

- *The software design is an iterative process through which requirements are translated into a **blueprint** for constructing the software.*

- Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design. Three characteristics that serve as a guide for the evaluation of a good design.

    - The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

    - The design must be a readable, understandable guide for those how generate code and for those who test and subsequently support the software.

    - The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains form an implementation perspective.

■ Quality Guidelines

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

- A design should exhibit an architecture that
  (a) has been created using recognizable architectural styles or patterns. (b) is composed of components that exhibit good design characteristics. (c) can be implemented in an evolutionary fashion, thereby facilitating

  implementation and testing.

- A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

■Quality Attributes

Hewlett Packard developed set of software quality attributes name FURPS. Functionality, Usability, Reliability, Performance and Supportability. The FURPS quality attributes represent a target for all software design.

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

- *Usability* is assessed by considering human factors, overall aesthetics, consistency and documentation.

- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure and the predictability of the program.

- *Performance* is measuring by processing speed, response time, resource consumption, throughput and efficiency.

- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability-these three attributes represent amore common term, maintainability-in addition, testability, compatibility, and configurability.

7.2 DESIGN CONCEPTS

Fundamental software design concepts provide the necessary frame work for "getting it right."

- Abstraction - Many levels of abstraction can be posed.
  - The highest level states the solution in broad terms using the language of the problem environment.
  - The lower level gives more detailed description of the solution.
  - *Procedural Abstraction* refers to a sequence of instruction that have specific and limited functions.
  - *Data Abstraction* is a named collection of data that describes a data object.

- Architecture – provides overall structure of the software and the ways in which the structure provides conceptual integrity for a system. The goal of software design is to derive and architectural rendering of a system which serves as a framework from which more detailed design activities can be conducted.
  - The Architectural design is represented by the following models
    - *Structural Model* – Organized collection of program components.
    - *Framework Model* – Increases level of design abstraction by identifying repeatable architectural design frameworks that are encountered in similar types of applications.
    - *Dynamic Model* – Addresses the behavioral aspects of the program architecture.
    - *Process Model* – Focuses on design of business or technical process that system must accommodate.
    - *Functional Model* – Used to represent functional hierarchy of the system.

- Patterns – A pattern is a named nugget of insight which conveys the essences of proven solutions to a recurring problem with a certain context amidst competing concerns. The design pattern provides a description that enables a designer to determine

    - Whether the pattern is applicable to the current work?
    - Whether the pattern can be reused?
    - Whether the pattern can serve as a guide for developing a similar but functionally or structurally different pattern?

- Modularity – the software is divided into separately named and addressable components called modules that are integrated to satisfy problem requirements.

- Information Hiding – modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information. Information hiding provides greatest benefits when modifications are required during testing and software maintenance.

- Functional Independence – is achieved by developing modules with single minded function and an aversion to excessive interaction with other modules.
  Independence is assessed using two qualitative criteria

    - ***Cohesion*** – is a natural extension of information hiding concept, module performs single task, requiring little interaction with other components in other parts of a program.

    - ***Coupling*** – interconnection among modules and a software structures, depends on interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

- Refinement – is a process of elaboration. Refinement causes the designer to elaborate on original statement, providing more and more detailed as each successive refinement occurs.

- Refactoring – important design activity suggested for agile methods, refactoring is a recognization technique that simplifies design of a component without changing its function or behavior. “Refactoring is a process of changing a software system in such away that it does not alter the external behavior of the code yet improves its internal structure.

- Design Classes – describes some element of problem domain, focusing on aspects of the problem that are user or customer visible. The software team must define a set of design classes that
  (1) Refine the analysis classes by providing design detail that will enable the classes to be implemented and

(2) Create a new set of design classes that implement a software infrastructure to support the business solution. Five different types of design classes each representing a different layer of the design architecture is suggested

- *User Interface classes* define all abstractions that are necessary for Human Computer Interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine) and the design classes for the interface may be visual representations of the elements of the metaphor.

- *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

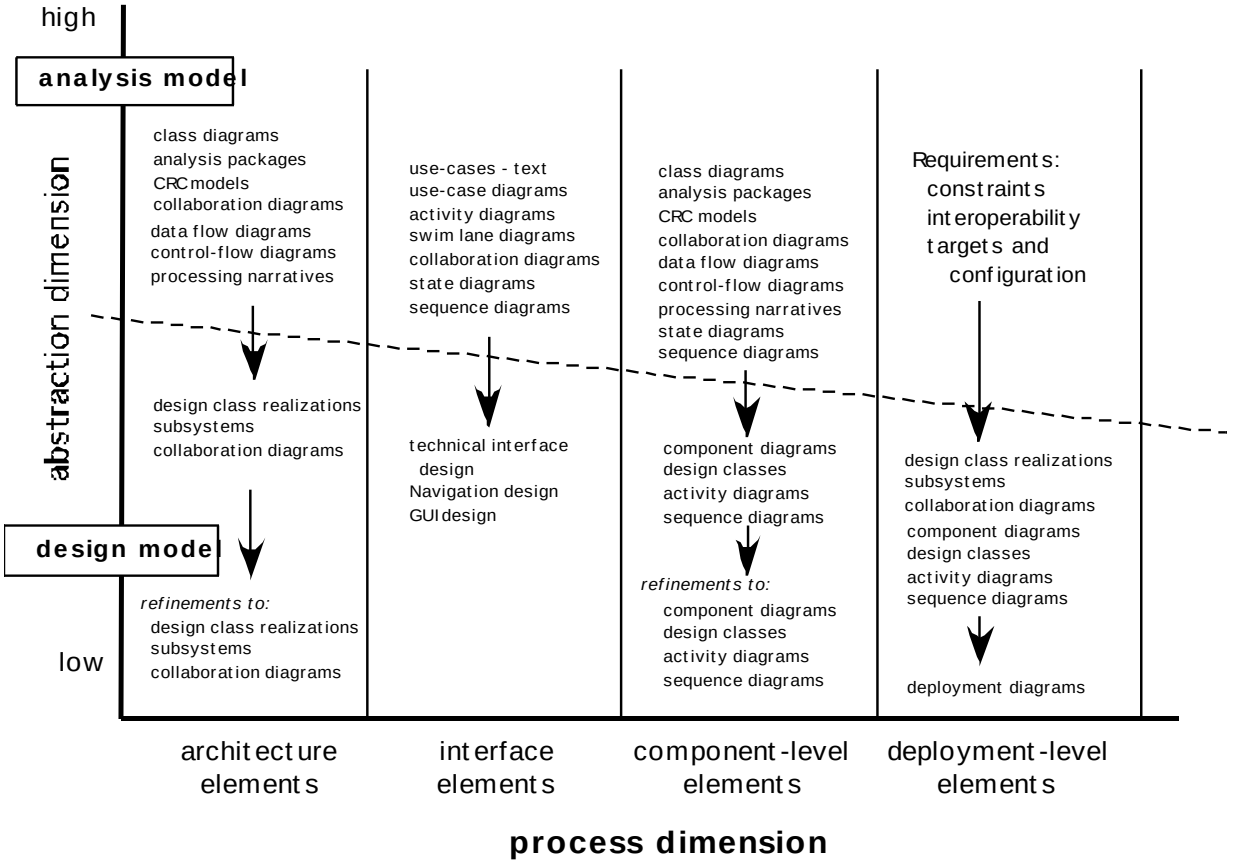
- *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.

- *Persistent classes* represent data stores (e.g, a database) that will persist beyond the execution of the software.

- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

7.3 THE DESIGN MODEL

- The design model can be viewed in two in two different dimensions. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.



■ DATA DESIGN ELEMENTS

- Like other software engineering activities, data design creates a model of data and/or information that is represented at a high level of abstraction.
- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.
- At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system.
- At the business level, the collection of information stored in disparate databases and reorganized into a “data warehouse” enables data mining or knowledge discovery that can have an impact on the success of the business itself.

■ ARCHITECTURAL DESIGN ELEMENTS

- The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms, their size, shape, and relationship to one another, and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software.

- The architectural model is derived from three sources:

  (1) Information about the application domain for the software to be built;

  (2) Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand

  (3) the availability of architectural patterns and styles.

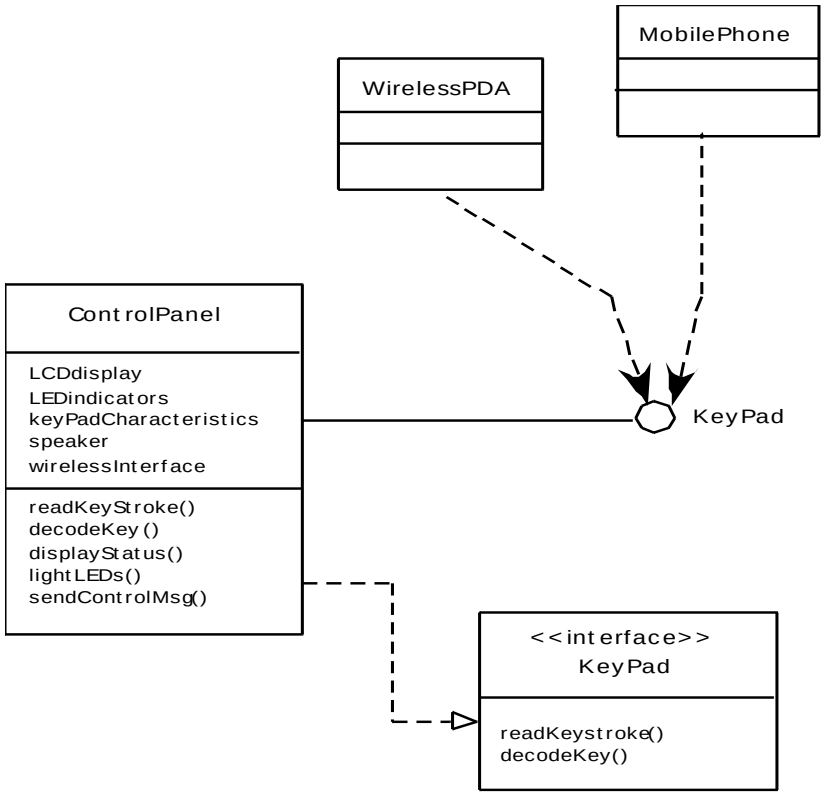
■ INTERFACE DESIGN ELEMENTS

- The interface design for software is the equivalent to a set of detailed drawing for the doors, windows and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utilities connections (e.g., water, electrical, gas, and telephone) come into the house and are distributed among the rooms depicted in the floor plan.

- There are three important elements of interface design:
  (1) the user interface (UI)

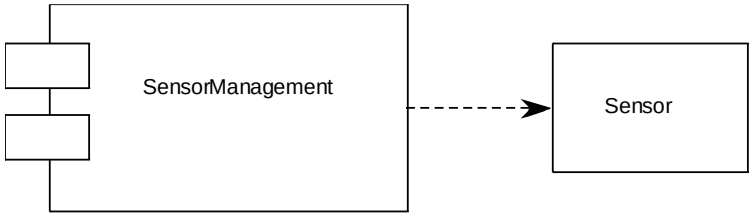
  (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and

  (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.



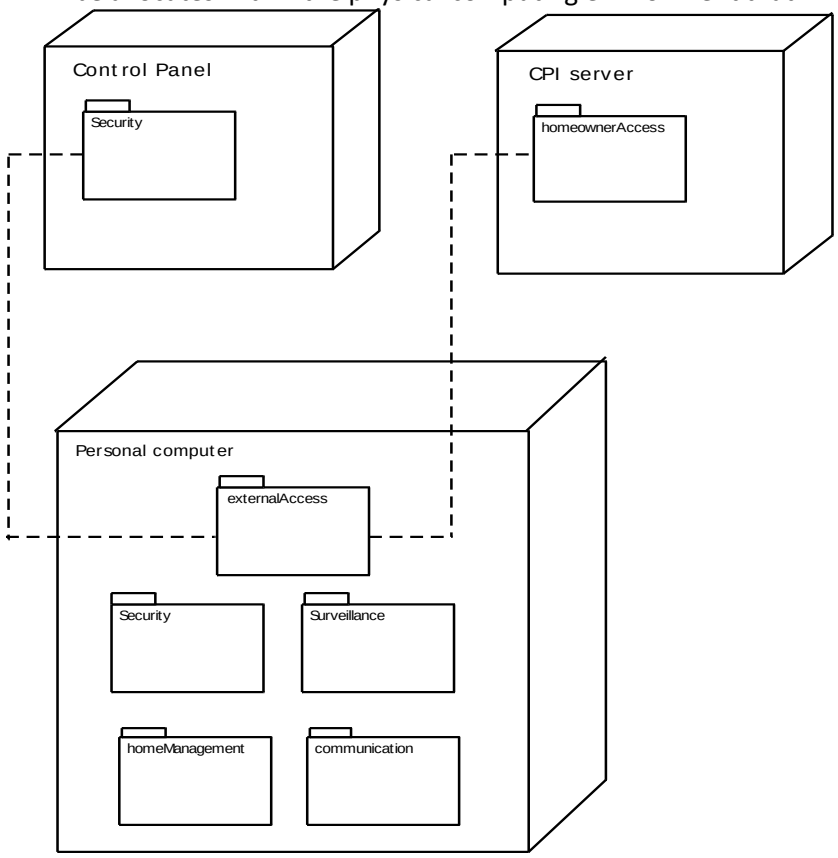
■ COMPONENT-LEVEL DESIGN ELEMENTS

- The component-level design for software is equivalent to a set of detailed drawings for each room in a house. These drawing depict wiring and plumbing within each room, the location of electrical receptacles and switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. The component-level design for software fully describes the internal detail of each software component.



■ DEPLOYMENT-LEVEL DESIGN ELEMENTS

- Deployment level design elements indicates how software functionality and subsystems will be allocates within the physical computing environment that will support the software.



---

8. CREATING AN ARCHITECTURAL DESIGN

- Design is an activity concerned with making major decisions, often of s structural nature.
- It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes.
- Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.

- *What is Architectural design?*
- *Who does it?*
- *Why is it important?*
- *What are the steps?*
- *What is the work product?*
- *How do I ensure that I have done it right?*

8.1 SOFTWARE ARCHITECTURE

- Effective software architecture and its explicit representation and design have become dominant themes in software engineering.

■ **Why Architecture?**

- The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

  (1) analyze the effectiveness of the design in meeting its stated requirements,

  (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

  (3) reduce the risks associated with the construction of the software.

■ **Why is Architecture Important?**

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

8.2 DATA DESIGN

The data design action translates data objects defined as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

■ **At the architectural level …**

- Design of one or more databases to support the application architecture
- Design of methods for 'mining' the content of multiple databases
  - navigate through existing databases in an attempt to extract appropriate business-level information

- Design of a data warehouse—a large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business

■ **At the component level …**

- refine data objects and develop a set of data abstractions
- implement data object attributes as one or more data structures
- review data structures to ensure that appropriate relationships have been established
- simplify data structures as required

- The systematic analysis principles applied to function and behavior should also be applied to data.
- All data structures and the operations to be performed on each should be identified.
- A data dictionary should be established and used to define both data and program design.
- Low level data design decisions should be deferred until late in the design process.
- The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
- A library of useful data structures and the operations that may be applied to them should be developed.
- A software design and programming language should support the specification and realization of abstract data types.

8.3 ARCHITECTURAL STYLES

- Each style describes a system category that encompasses:
  (1) a set of components (e.g., a database, computational modules) that perform a function required by a system,

  (2) a set of connectors that enable "communication, coordination and cooperation" among components,

  (3) constraints that define how components can be integrated to form the system, and

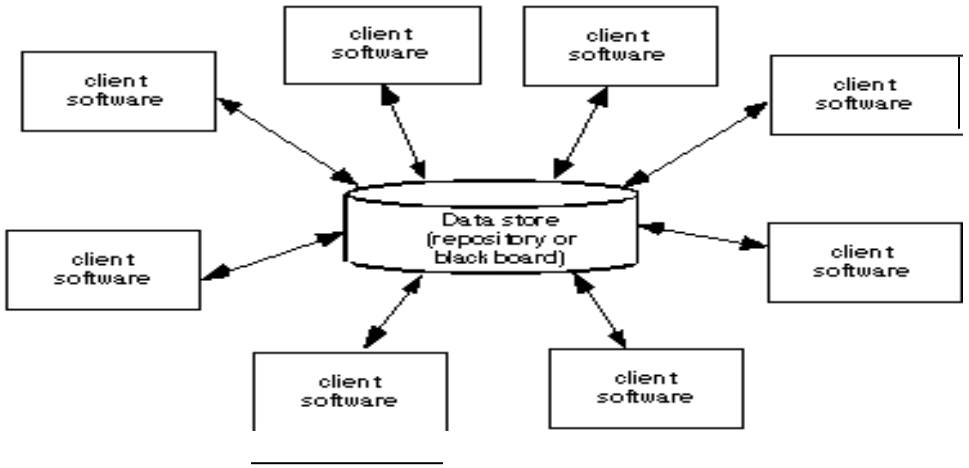
  (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

- Data-centered architectures

- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

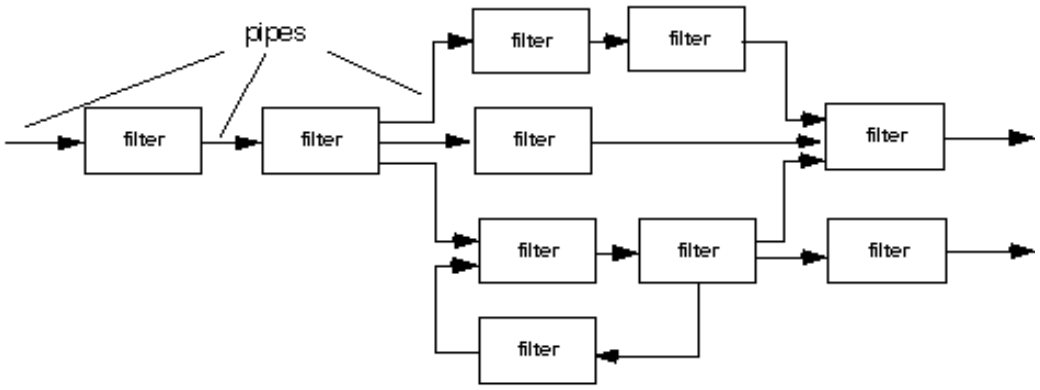
■ **Data-Centered Architecture**

- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. The following figure illustrates a typical data-centered style.

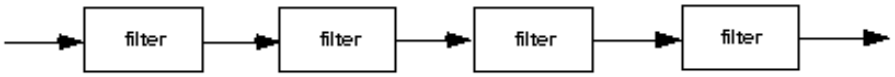


■ **Data Flow Architecture**

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe and filter structure has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential.



(a) pipes and filters



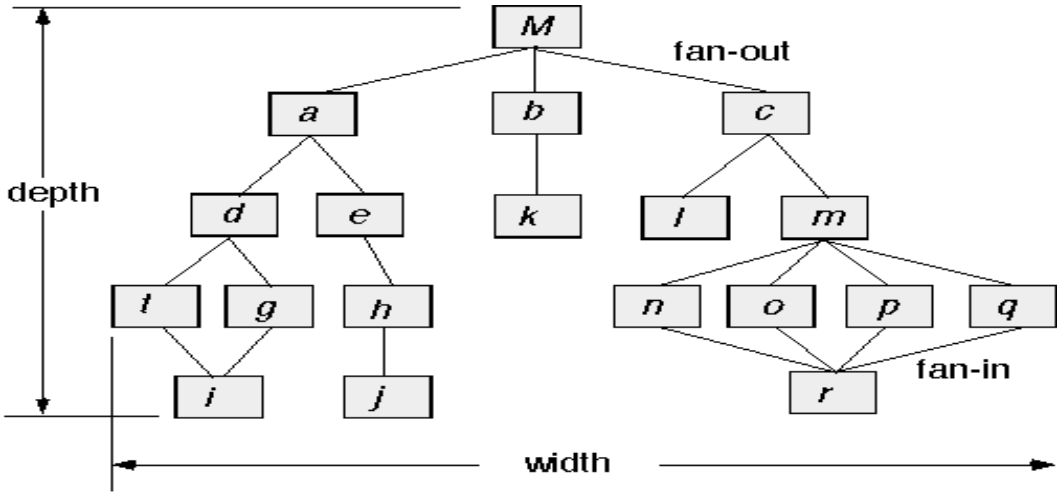
(b) batch sequential

■ **Call and Return Architecture**

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. Two substyles exist within this category:

    - *Main program/subprogram architecture.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. The following figure illustrates architecture of this type.

    - *Remote procedure call architecture.* The components of main program /subprogram architecture are distributed across multiple computers on a network.

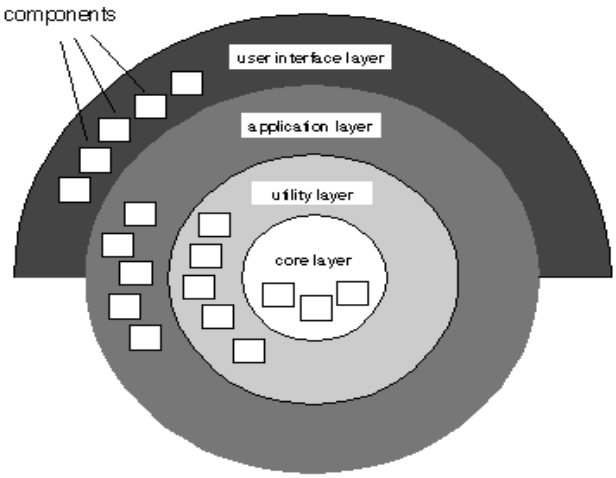


■ **Object-oriented architecture**

- The component of a system encapsulates data and the operations that must be applied to manipulate the data communication and coordination between components is accomplished via message passing.

■ **Layered Architecture**

- The basic structure of a layered architecture is illustrated in the following figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



8.4 ARCHITECTURAL PATTERNS

- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
    - *operating system process management* pattern
    - *task scheduler* pattern

- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
    - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
    - an *application level persistence* pattern that builds persistence features into the application architecture

- Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment
    - A *broker* acts as a 'middle-man' between the client component and a server component.

8.5 ARCHITECTURAL DESIGN

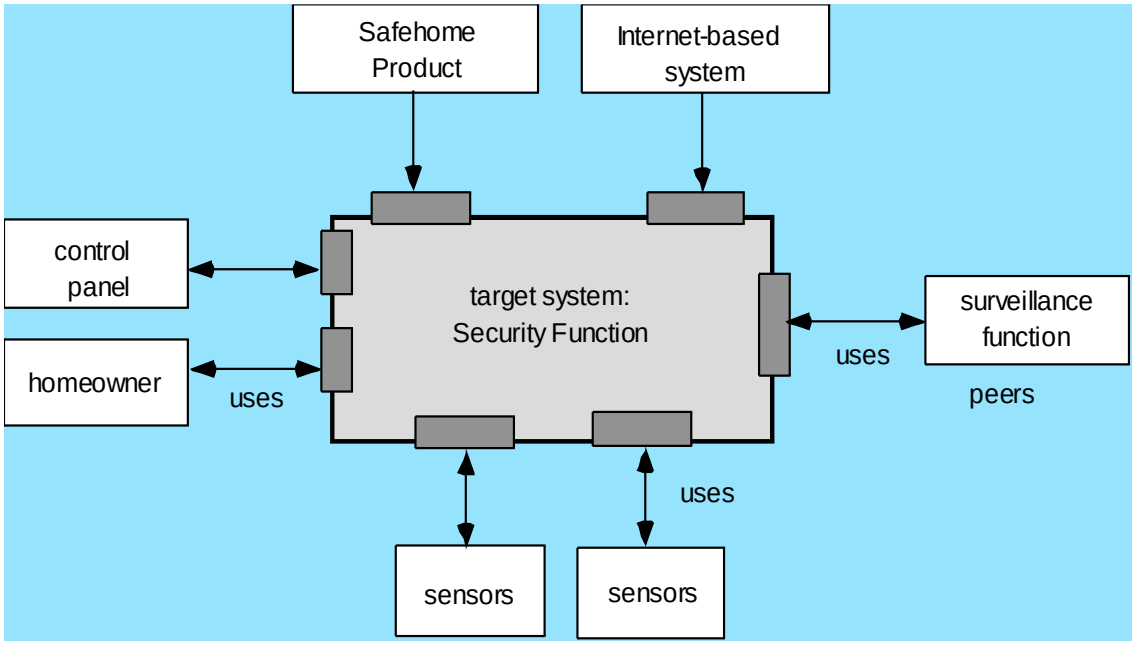
- The software must be placed into context
    - the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction

- A set of architectural archetypes should be identified
    - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior

- The designer specifies the structure of the system by defining and refining software components that implement each archetype

■ Architectural Context

- A system engineer must model context. A system context diagram accomplishes this requirement by representing the flow of information into and out of the system, the user interface and relevant support processing. At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure.



■ Archetypes

- the *SafeHome* home security function, we might define the following archetypes:

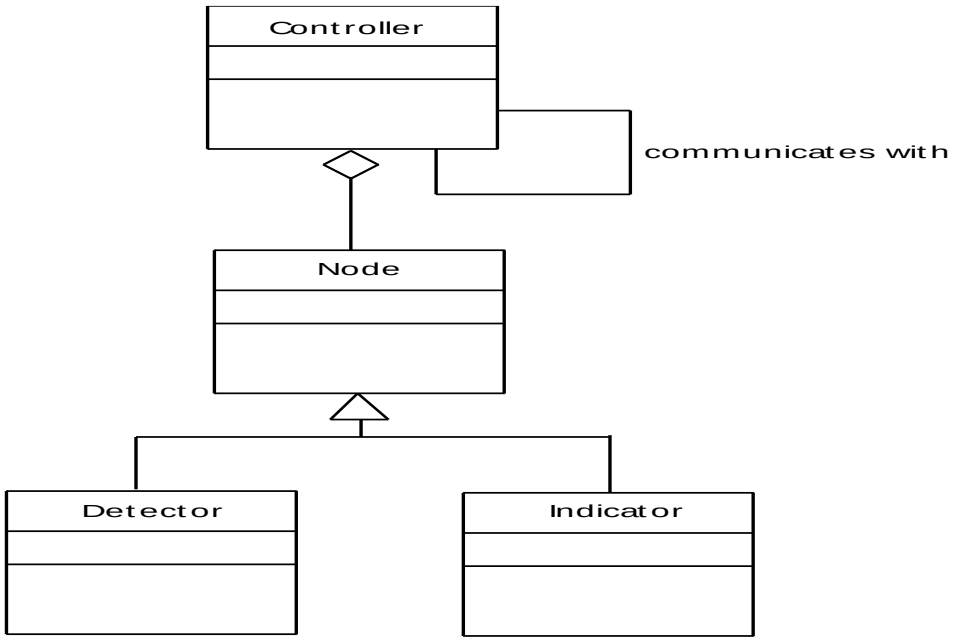
    - Node – Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of
      (1) various sensors and

(2) a variety of alarm (output) indicators.

- Dectector – An abstraction that encompasses all sensing equipment that feeds information into the target system.

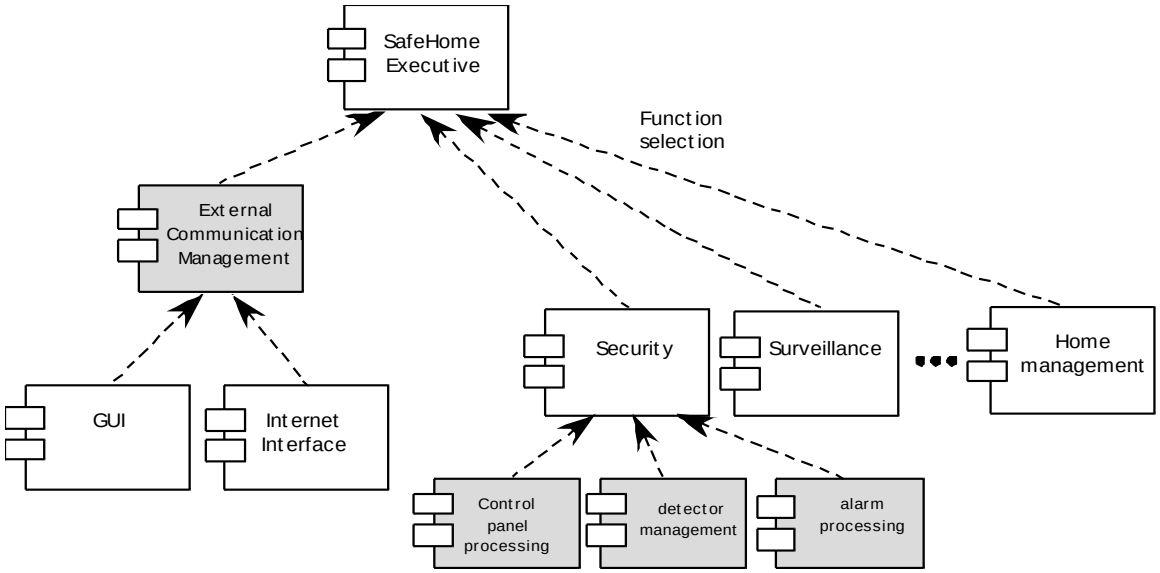
- Indicator – An abstraction that represents all mechanism (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

- Controller – An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.



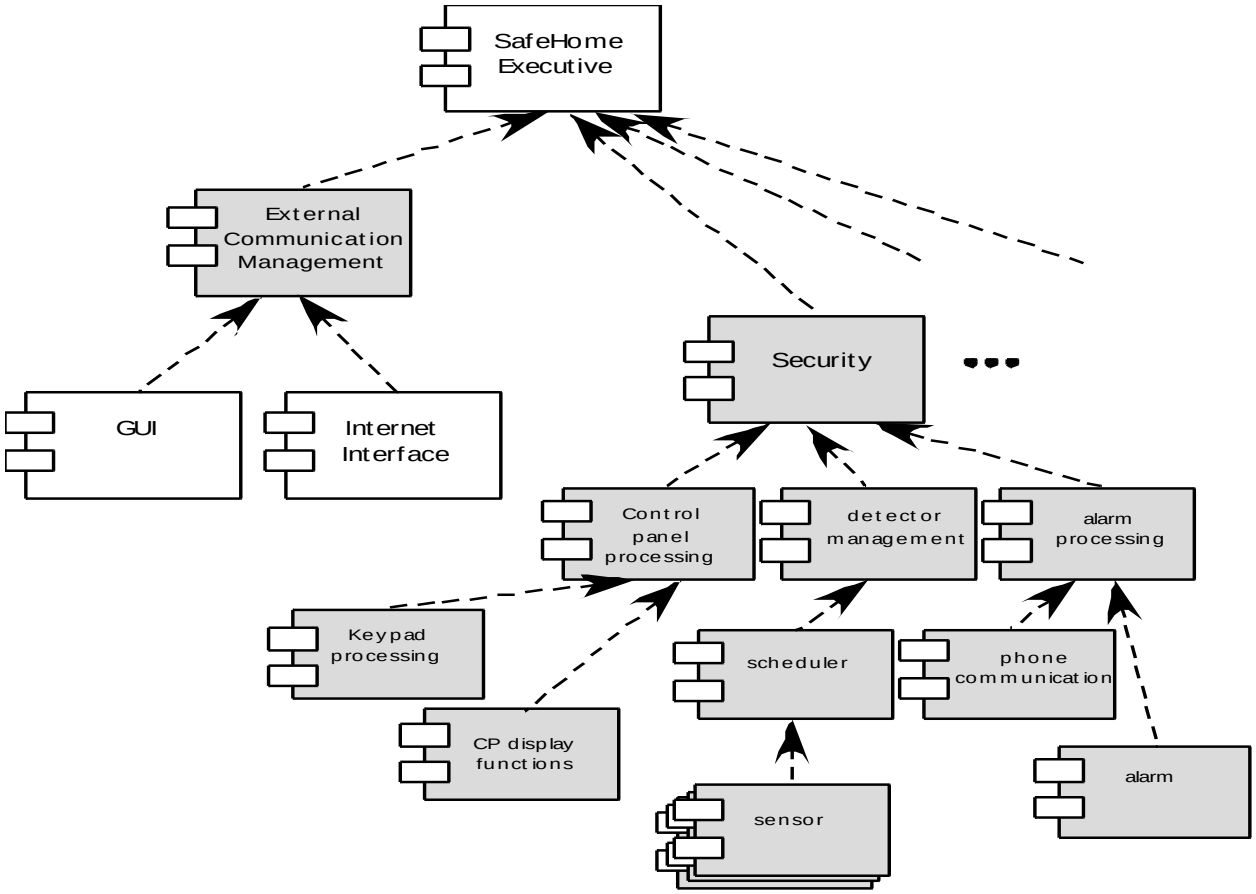
■ Refining the Architecture into Components

- Continuing the *SafeHome* home security function example, we might define the set of top-level components that address the following functionality.
  - *External communication management* – coordinates communication of the security function with external entities, for example, internet-based systems. External alarm notification.
  - *Control panel processing* – manages all control panel functionality.
  - *Detector management* – coordinates access to all detectors attached to the system.
  - *Alarm processing* – verifies and acts on all alarm conditions.



■ **Describing Instantiations of the System**

- The instantiation of the architecture means that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.
    - Fig Illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in figure are refined further to show additional detail.


SafeHome Executive
External Communication Management
Security
GUI
Internet Interface
Control panel processing
detector management
alarm processing
Keypad processing
scheduler
phone communication
CP display functions
alarm
sensor

9. OBJECT-ORIENTED DESIGN

OBJECTIVES

- To explain how a software design may be represented as a set of interacting objects that manage their own state and operations.
- To describe the activities in the object-oriented design process.
- To introduce various models that can be used to describe an object-oriented design.
- To show how the UML may be used to represent these models.
- Not about implementation, but about design.

■ TOPICS COVERED

- Objects and object classes
- An object-oriented design process
- Design evolution

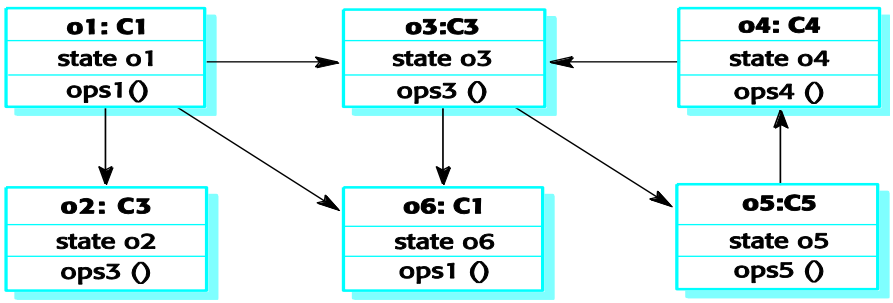
■ OBJECT-ORIENTED DEVELOPMENT

- Object-oriented analysis, design and programming are related but distinct.
- OOA is concerned with developing an object model of the application domain.
- OOD is concerned with developing an object-oriented system model to implement requirements.
- OOP is concerned with realising an OOD using an OO programming language such as Java or C++.

■ Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated. Objects communicate by message passing (no global variables).
- Objects may be distributed and may execute sequentially or in parallel.

■ Interacting objects



■ **Advantages of OOD**

- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

9.1 OBJECTS AND OBJECT CLASSES

- Objects are entities in a software system which represents instances of real-world and system entities.
- Object classes are templates for objects. They may be used to create objects.
- Object classes may inherit attributes and services from other object classes.

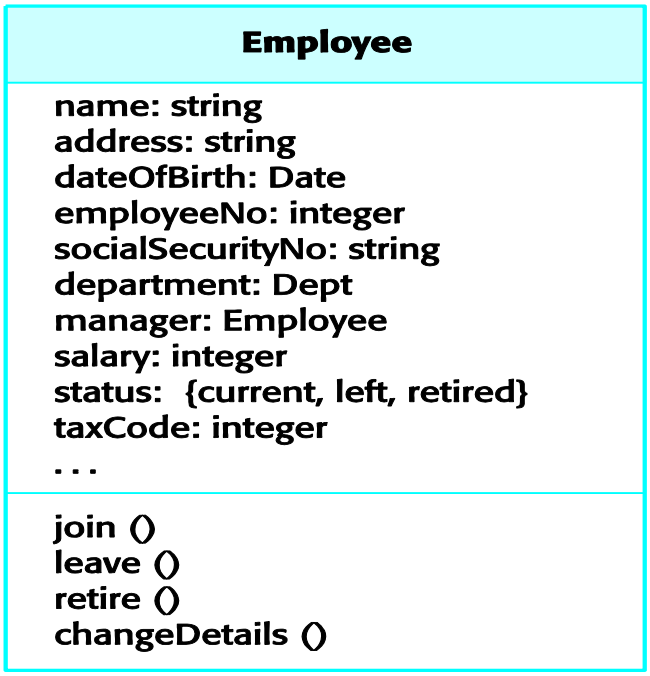
An object is an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

Objects are created according to some object class definition. An object class definition serves as a template for objects. It includes declarations of all the attributes and services which should be associated with an object of that class.

■ **The Unified Modeling Language**

- Several different notations for describing object-oriented designs were proposed in the 1980s and 1990s.
- The Unified Modeling Language is an integration of these notations.
- It describes notations for a number of different models that may be produced during OO analysis and design.
- It is now a *de facto* standard for OO modelling.

■ **Employee object class (UML)**



■ **Object communication**

- Conceptually, objects communicate by message passing.
- Messages
  - The name of the service requested by the calling object;
  - Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
  - Name = procedure name;
  - Information = parameter list.

■ **Message examples**

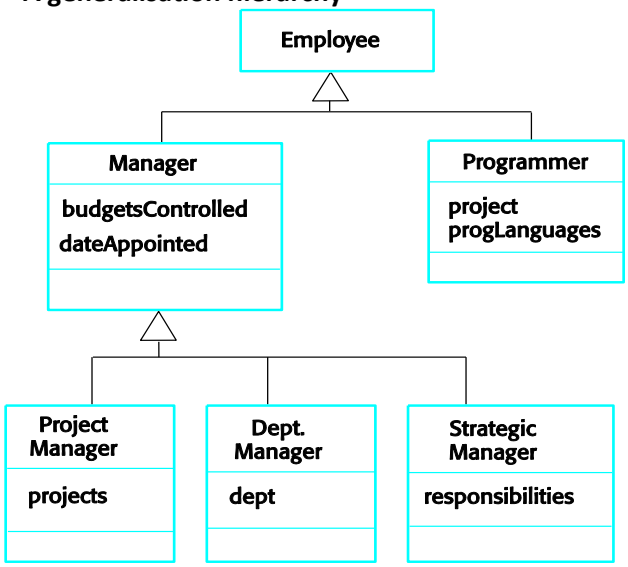
```
Call a method associated with a buffer
// object that returns the next value
//   in the buffer

                    v = circularBuffer.Get () ;
```

```
                    // Call the method associated with a
               // thermostat object that sets the
               // temperature to be maintained

                                   thermostat.setTemp (20) ;
```

■ Generalisation and inheritance

- Objects are members of classes that define attribute types and operations.
- Classes may be arranged in a class hierarchy where one class (a super-class) is a generalisation of one or more other classes (sub-classes).
- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.
- Generalisation in the UML is implemented as inheritance in OO programming languages.

■ A generalisation hierarchy



■ Advantages of inheritance

- It is an abstraction mechanism which may be used to classify entities.
- It is a reuse mechanism at both the design and the programming level.
- The inheritance graph is a source of organisational knowledge about domains and systems.

■ Problems with inheritance

- Object classes are not self-contained. They cannot be understood without reference to their super-classes.
- The inheritance graphs of analysis, design and implementation have different functions and should be separately maintained.

9.2 AN OBJECT-ORIENTED DESIGN PROCESS

- Structured design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an essential communication mechanism.

■ **Process stages**

- Highlights key activities
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

■ **Weather system description**

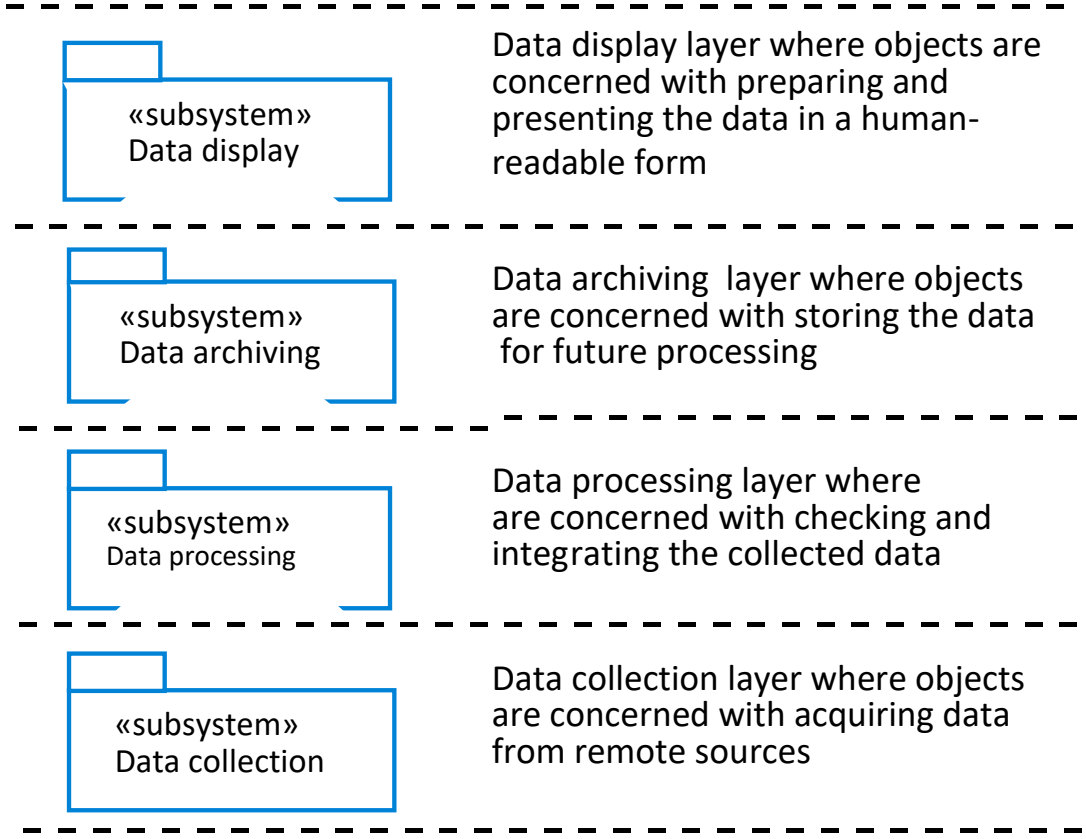
- A weather mapping system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites. Weather stations transmit their data to the area computer in response to a request from that machine.

- The area computer system validates the collected data and integrates it with the data from different sources. The integrated data is archived and, using data from this archive and a digitised map database a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.

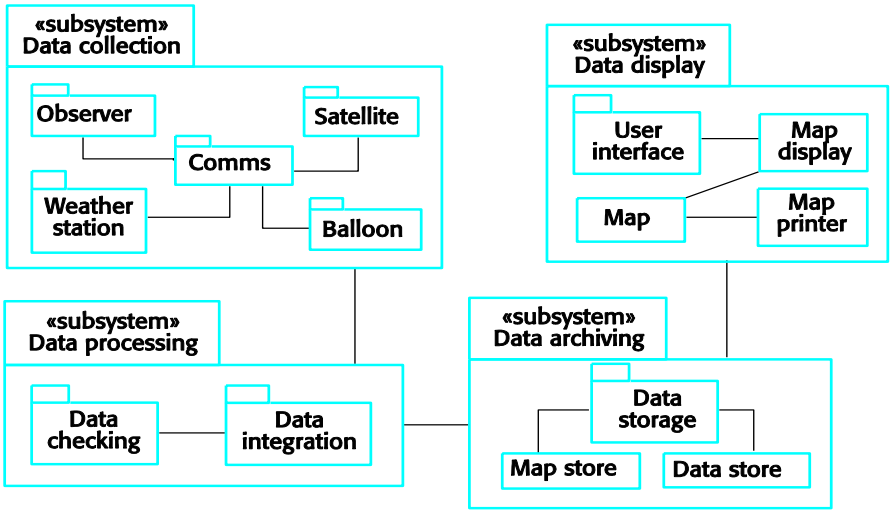
■ **System context and models of use**

- Develop an understanding of the relationships between the software being designed and its external environment
- System context
  - A static model that describes other systems in the environment. Use a subsystem model to show other systems. Following slide shows the systems around the weather station system.
- Model of system use
  - A dynamic model that describes how the system interacts with its environment. Use use-cases to show interactions

■ **Layered architecture**



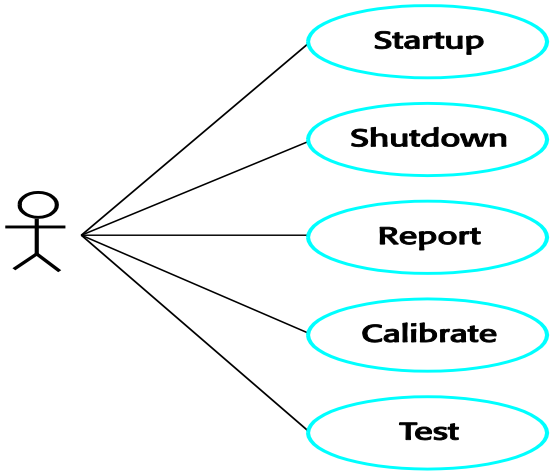
■ **Subsystems in the weather mapping system**



■ **Use-case models**

- Use-case models are used to represent each interaction with the system.
- A use-case model shows the system features as ellipses and the interacting entity as a stick figure.

▪ **Use-cases for the weather station**



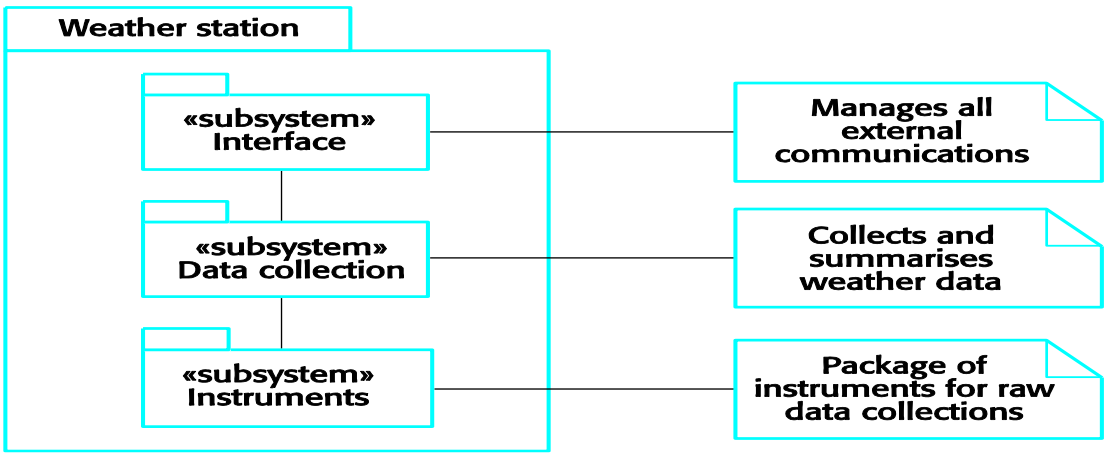
▪ **Use-case description**

System	Weather station
Use-case	Report
Actors	Weather data collection system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather data collection system. The data sent are the maximum minimum and average ground and air temperatures, the maximum, minimum and average air pressures, the maximum, minimum and average wind speeds, the total rainfall and the wind direction as sampled at 5 minute intervals.
Stimulus	The weather data collection system establishes a modem link with the weather station and requests transmission of the data.
Response	The summarised data is sent to the weather data collection system
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to the other and may be modified in future.

■ Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- A layered architecture as discussed in Chapter 11 is appropriate for the weather station
    - Interface layer for handling communications;
    - Data collection layer for managing instruments;
    - Instruments layer for collecting data.
- There should normally be no more than 7 entities in an architectural model.

■ Weather station architecture



■ Object identification

- Identifying objects (or object classes) is the most difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

■ Approaches to identification

- Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).
- Base the identification on tangible things in the application domain.
- Use a behavioural approach and identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

■ Weather station description

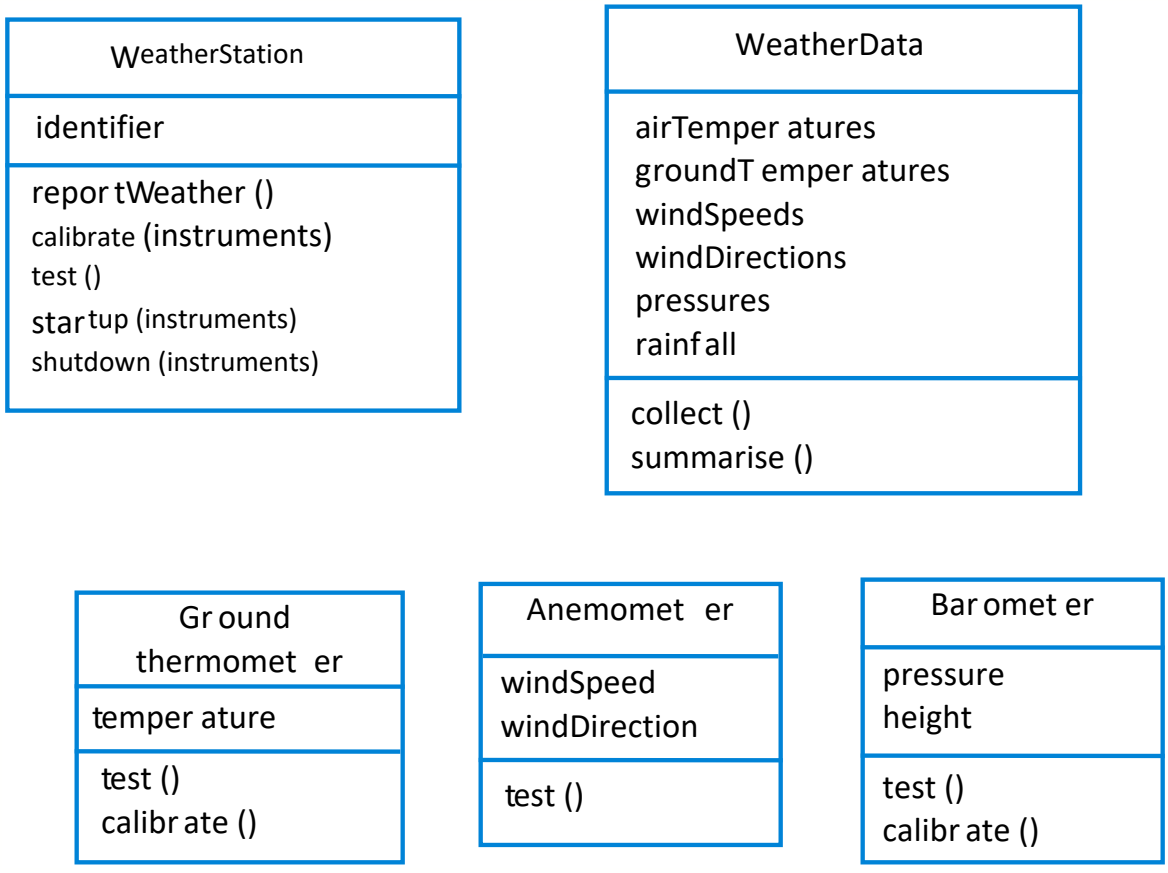
- Weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.

- When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

■ **Weather station object classes**

- Ground thermometer, Anemometer, Barometer
  - Application domain objects that are 'hardware' objects related to the instruments in the system.
- Weather station
  - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
- Weather data
  - Encapsulates the summarised data from the instruments.

■ **Weather station object classes**



■ Further objects and object refinement

- Use domain knowledge to identify more objects and operations
    - Weather stations should have a unique identifier;
    - Weather stations are remotely situated so instrument failures have to be reported automatically. Therefore attributes and operations for self-checking are required.

■ Design models

- Design models show the objects and object classes and relationships between these entities.
- Static models describe the static structure of the system in terms of object classes and relationships.
- Dynamic models describe the dynamic interactions between objects.

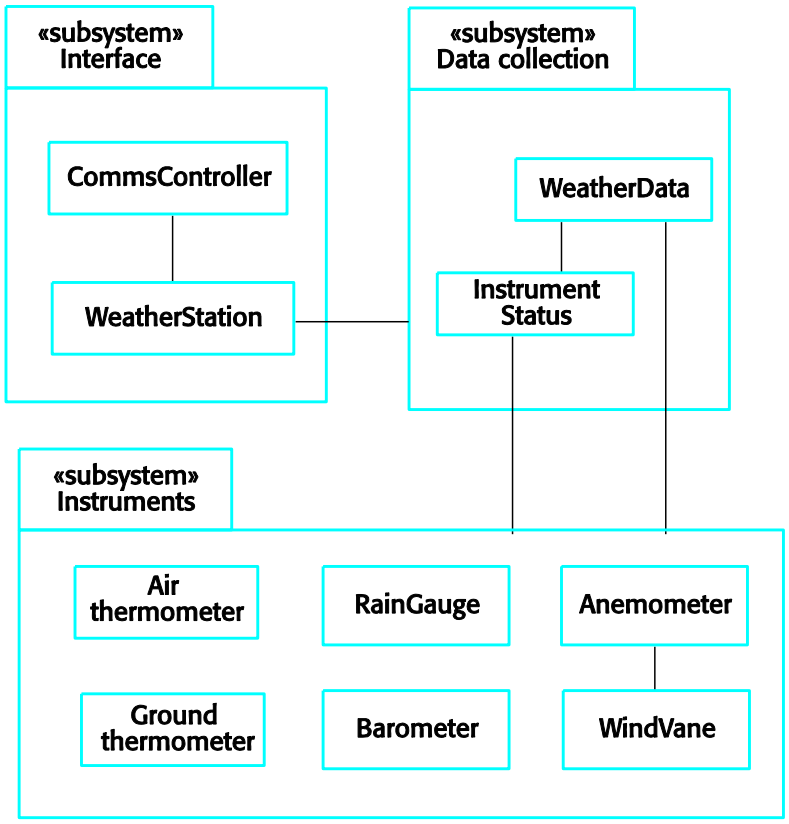
■ Examples of design models

- Sub-system models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

■ Subsystem models

- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

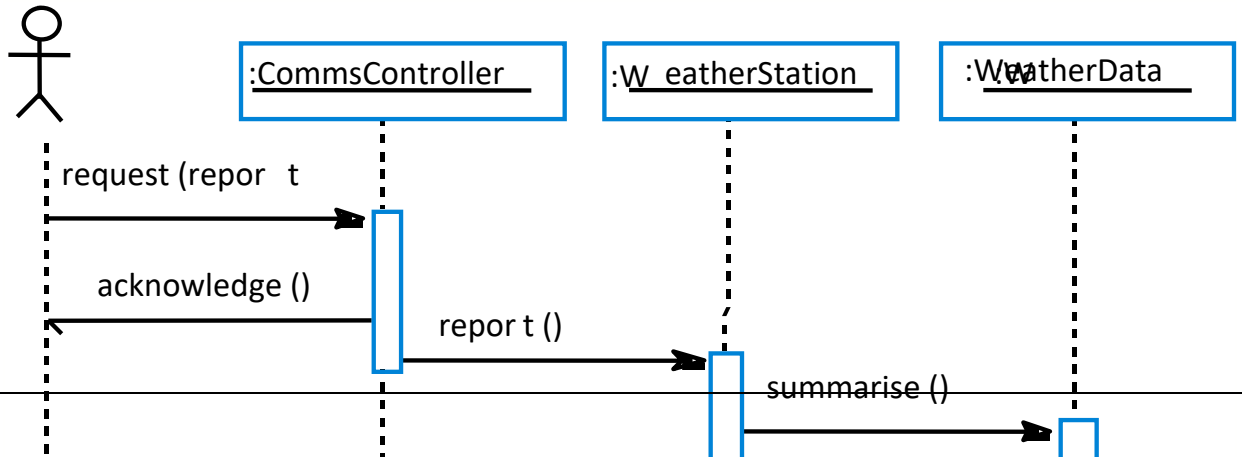
■ Weather station subsystems



■ **Sequence models**

- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.
  -

■ **Data collection sequence**

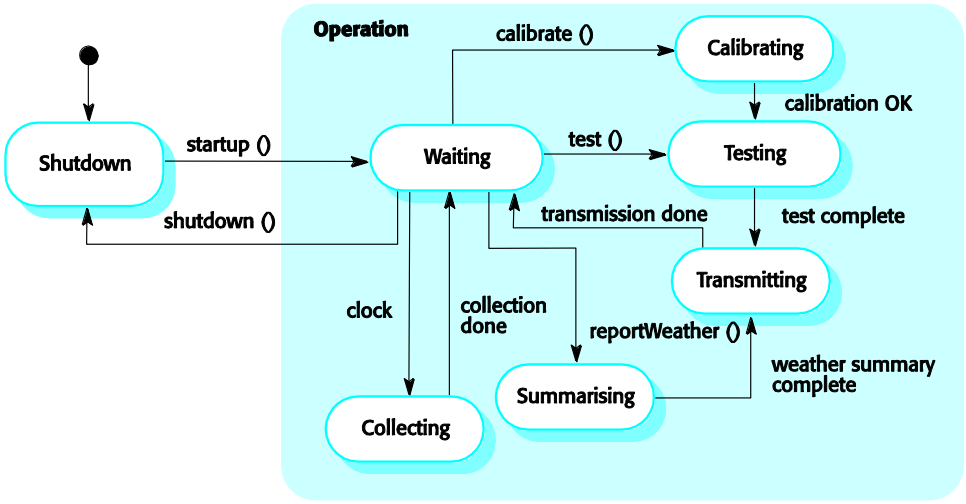


■ Statecharts

- Show how objects respond to different service requests and the state transitions triggered by these requests

    - If object state is Shutdown then it responds to a Startup() message;
    - In the waiting state the object is waiting for further messages;
    - If reportWeather () then system moves to summarising state;
    - If calibrate () the system moves to a calibrating state;
    - A collecting state is entered when a clock signal is received.

■ Weather station state diagram



Object interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

- **Weather station interface**

```
interface WeatherStation {

        public void WeatherStation () ;

        public void startup () ;
        public void startup (Instrument i) ;

        public void shutdown () ;
        public void shutdown (Instrument i) ;

        public void reportWeather ( ) ;

        public void test () ;
        public void test ( Instrument i ) ;

        public void calibrate ( Instrument i) ;

        public int getID () ;

} //WeatherStation
```

9.3 DESIGN EVOLUTION

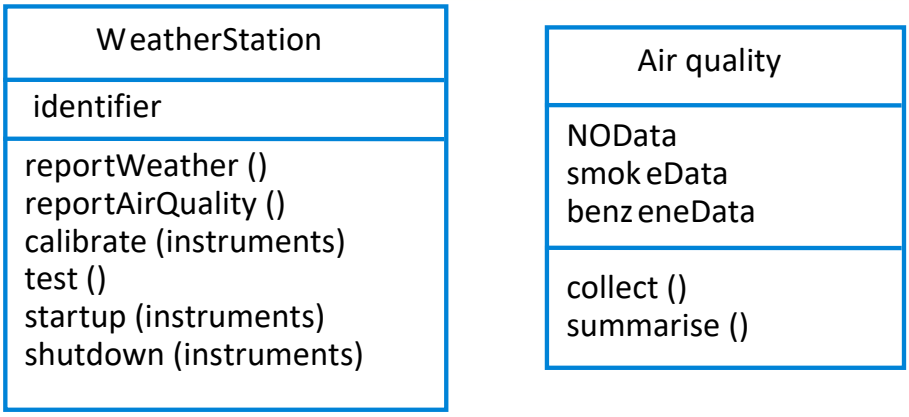
- Hiding information inside objects means that changes made to an object do not affect other objects in an unpredictable way.

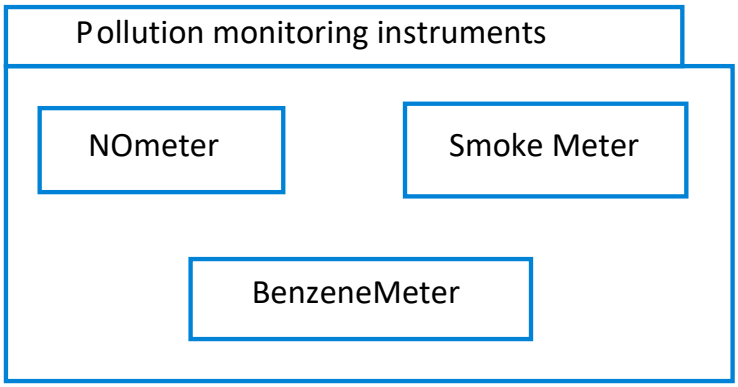
- Assume pollution monitoring facilities are to be added to weather stations. These sample the air and compute the amount of different
  pollutants in the atmosphere.
- Pollution readings are transmitted with weather data.

■ **Changes required**

- Add an object class called Air quality as part of WeatherStation.
- Add an operation reportAirQuality to WeatherStation. Modify the control software to collect pollution readings.
- Add objects representing pollution monitoring instruments.

■ **Pollution monitoring**





10. PERFORMING USER INTERFACE DESIGN

- User Interface design create an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

- **Interface Design**

  Easy to learn?

  Easy to use?

  Easy to understand?

- ***Typical Design Errors***

  lack of consistency

  too much memorization

  no guidance / help

  no context sensitivity

  poor response

  Arcane/unfriendly

10.1 GOLDEN RULES

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

- **Place the User in Control**
  - Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
  - Provide for flexible interaction.
  - Allow user interaction to be interruptible and undoable.
  - Streamline interaction as skill levels advance and allow the interaction to be customized.
  - Hide technical internals from the casual user.
  - Design for direct interaction with objects that appear on the screen.

- **Reduce the User's Memory Load**
  - Reduce demand on short-term memory.
  - Establish meaningful defaults.
  - Define shortcuts that are intuitive.
  - The visual layout of the interface should be based on a real world metaphor.
  - Disclose information in a progressive fashion.

- **Make the Interface Consistent**
  - Allow the user to put the current task into a meaningful context.
  - Maintain consistency across a family of applications.
  - If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

10.2 USER INTERFACE ANALYSIS AND DESIGN

- The overall process for analyzing and designing a user interface begins with the creation of different models of system function.

■ **Interface Analysis and Design Models**

- User model — a profile of all end users of the system

- Design model — a design realization of the user model

- Mental model (system perception) — the user's mental image of what the interface is
- Implementation model — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics.

■ **User Interface Design Process**

- The analysis and design process for user interfaces is iterative and can be represented using a spiral model.

- The user interface analysis and design process encompasses four distinct framework activities.

  - User, task and environment analysis and modeling.
  - Interface design.
  - Interface construction (implementation).
  - Interface validation.

- The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are

    - Where will the interface be located physically?
    - Will the user be sitting, standing or performing other tasks unrelated to the interface?
    - Does the interface hardware accommodate space, light or noise constraints?
    - Are there special human factors consideration driven by environmental factors?

- The information gathered as part of the analysis activity is used to create an analysis model for the interface.

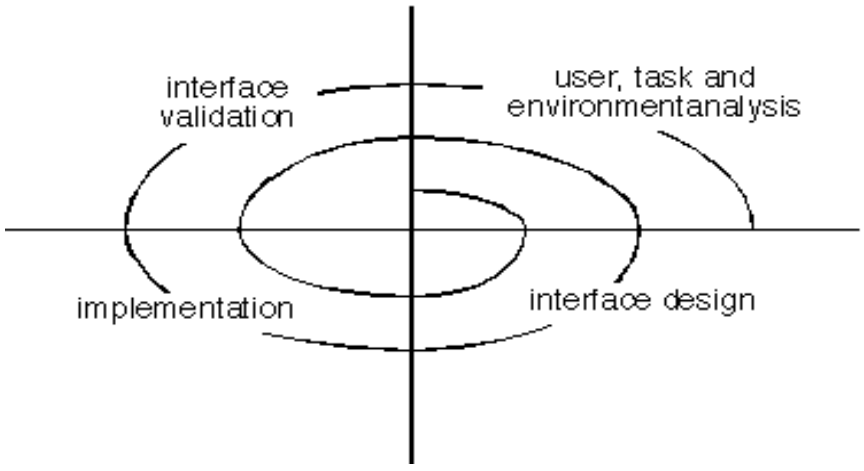
- The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

- Validation focuses on

    (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements.

    (2) the degree to which the interfaced is easy to use and easy to learn

    (3) the users acceptance of the interface as a useful tool in their work.



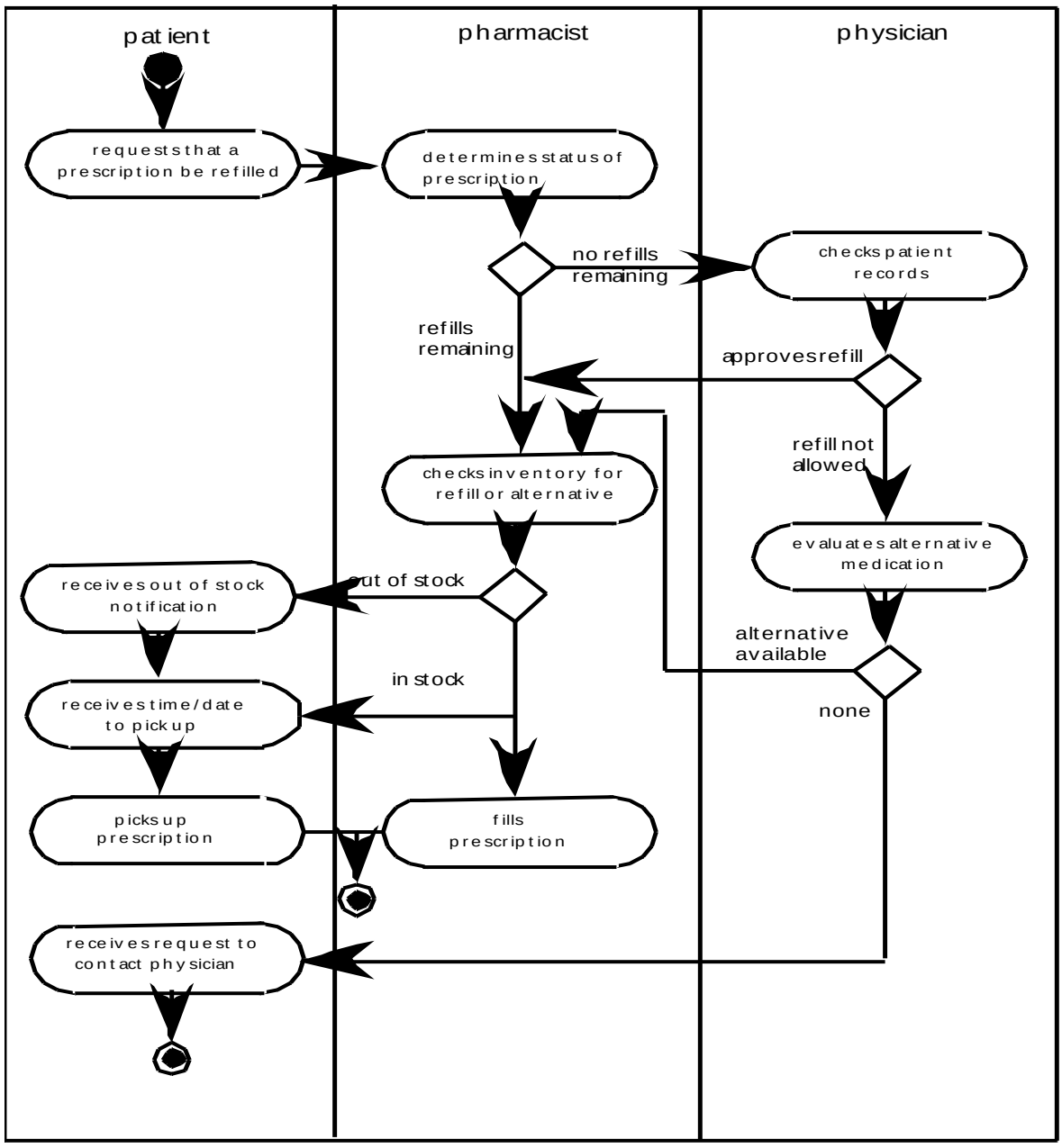
10.3 INTERFACE ANALYSIS

- Interface analysis means understanding
  - the people (end-users) who will interact with the system through the interface;
  - the tasks that end-users must perform to do their work,
  - the content that is presented as part of the interface
  - the environment in which these tasks will be conducted**.**

- **User Analysis**
  - Are users trained professionals, technician, clerical, or manufacturing workers?
  - What level of formal education does the average user have?
  - Are the users capable of learning from written materials or have they expressed a desire for classroom training?
  - Are users expert typists or keyboard phobic?
  - What is the age range of the user community?
  - Will the users be represented predominately by one gender?
  - How are users compensated for the work they perform?
  - Do users work normal office hours or do they work until the job is done?
  - Is the software to be an integral part of the work users do or will it be used only occasionally?
  - What is the primary spoken language among users?
  - What are the consequences if a user makes a mistake using the system?
  - Are users experts in the subject matter that is addressed by the system?
  - Do users want to know about the technology the sits behind the interface?

- **Task Analysis and Modeling**
  - Answers the following questions …
    - What work will the user perform in specific circumstances?
    - What tasks and subtasks will be performed as the user does the work?
    - What specific problem domain objects will the user manipulate as work is performed?
    - What is the sequence of work tasks—the workflow?
    - What is the hierarchy of tasks?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

- **Swimlane Diagram**



- **Analysis of Display Content**

  - Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
  - Can the user customize the screen location for content?

- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color to be used to enhance understanding?
- How will error messages and warning be presented to the user?

10.4 INTERFACE DESIGN STEPS

- Using information developed during interface analysis; define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
- Depict each interface state as it will actually look to the end-user.
- Indicate how the user interprets the state of the system from information provided through the interface.

■ Interface Design Patterns

The Design pattern is an abstraction that prescribes a design solution to a specific, well bounded design problem

- Patterns are available for
    - The complete UI
    - Page layout
    - Forms and input
    - Tables
    - Direct data manipulation
    - Navigation
    - Searching
    - Page elements
    - e-Commerce

■ Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

10.5 DESIGN EVALUATION

- Design evaluation determines whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provide impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.

- After the design model has been completed, a first level prototype is created.

- The design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews.

  - The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
  - The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
  - The number of actions, tasks and system states indicated by the design model imply the memory load on users of the system
  - Interface style, help facilities and error handling protocol provide indication of the complexity of the interface and the degree to which it will be accepted by the user.

- To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be

  (1) simple yes/no response,

  (2) numeric response,

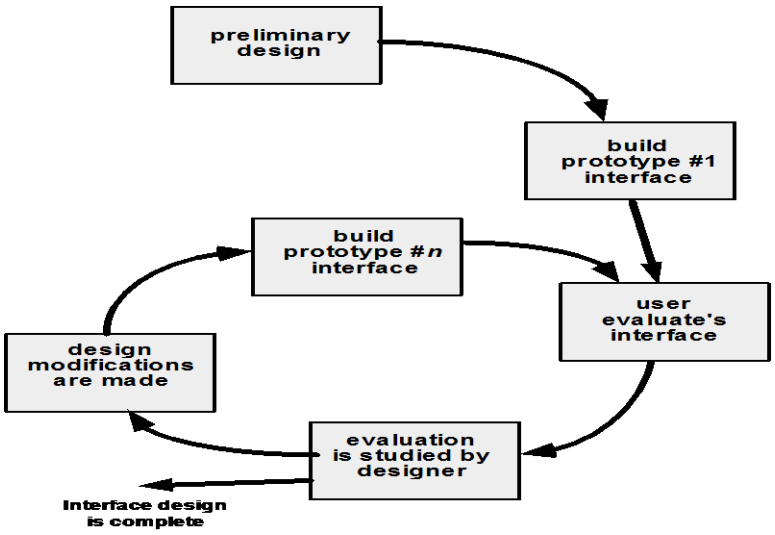
  (3) scaled (subjective) response,

  (4) likert scales (e.g., strongly agree, somewhat agree),

  (5) percentage (subjective) response or

  (6) open-ended.

- If quantitative data are desired, a form of time study analysis can be conducted.

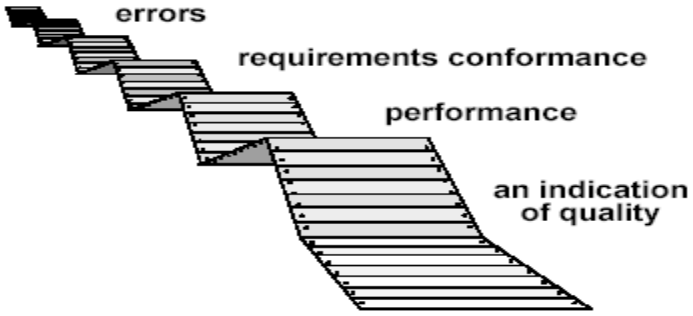


11. TESTING STRATEGIES

- **Software Testing**

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

- **What Testing Shows**



- **Who Tests the Software?**

developer

Understands the system
but, will test "gently"
and, is driven by "delivery"

independent tester

Must learn about the system,
but, will attempt to break it
and, is driven by quality

11.1 A STRATEGIC APPROACH TO SOFTWARE TESTING

- A number of software testing strategies have proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

  - To perform effective testing, a software team should conduct effective formal technical reviews. By doing this, many errors will be eliminated before testing commences.
  - Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
  - Different testing techniques are appropriate at different points in time.
  - Testing is conducted by the developer of the software and (for large projects) an independent test group.
  - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

- **Verification and Validation**

  - Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

    - Verification: Are we building the product right?
    - Validation: Are we building the right product?

  - Verification and validation encompasses a ;wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, and installation testing.

- **Organizing for software testing**

- The people who have built the software are now asked to test the software.
- Unfortunately, developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements, and that it will be completed on schedule and within budget.
- There are often a number of misconceptions

  (1) that the developer of software should do no testing at all,

  (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly,

  (3) that testers get involved with the project only when the testing steps are about to begin.

- In many cases, the developer also conducts integration testing-a testing step that leads to the construction of the complete software architecture. Only after the software architecture is complete does an independent test group become involved.
- The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built.
- The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

■ **A software Testing Strategy for conventional software**

- The module (component) is our initial focus
- Integration of modules follows

■ **A software Testing Strategy for Object Oriented software**

- our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

■ **Criteria for completion of testing**

- Question arises every time software testing is discussed: when are we done testing-how do we know that we've tested enough?
- One response to the question is: You're never done testing; the burden simply shifts from you to your customer. Every time the customer/user executes a computer program, the program is being tested.
- You're done testing when you run out of time or you run out of money.
- By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: when are we done testing?

■ STRATEGIC ISSUES

- State testing objectives explicitly.

- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective formal technical reviews as a filter prior to testing
- Conduct formal technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

11.2. TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

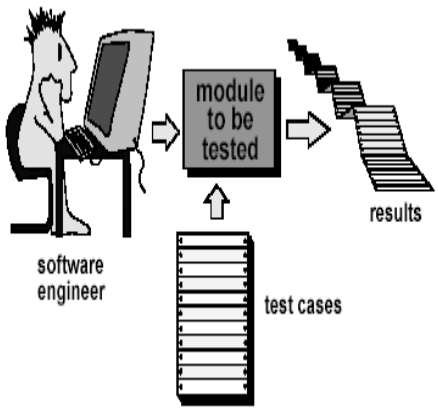
- There are many strategies that can be used to test software:

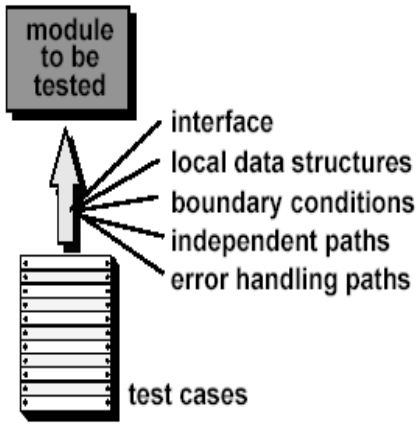
    - A software team could wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors. This approach, although appealing, simply does not work. It will result in buggy software that disappoints the customer and end-user.

    - A software engineer could conduct tests on a daily basis, whenever any part of the system is constructed. This approach, although less appealing to many, can be very effective.

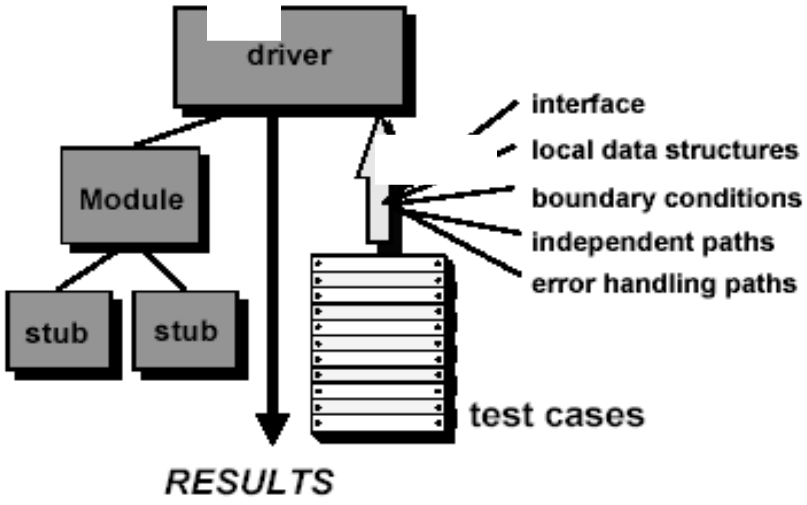
- **Unit Testing**

    - Unit testing focuses verification effort on the smallest unit of software design- the software component or module.



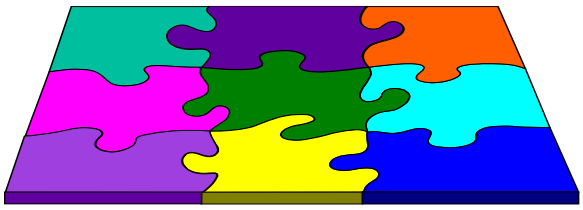


- **Unit Test Environment**



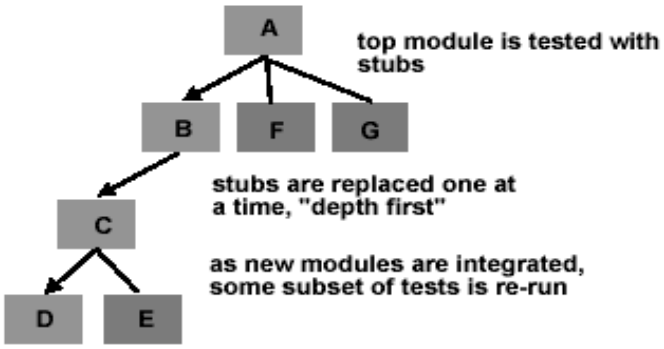
Integration Testing

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
  Options:
  - the “big bang” approach
  - an incremental construction strategy



Top Down Integration

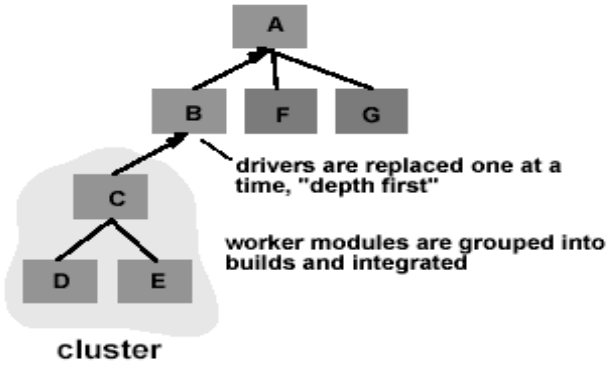
- Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module.



- The integration process is performed in a series of five steps.
  - The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.
  - Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
  - Tests are conducted as each component is integrated.
  - On completion of each set of tests, another stub is replaced with the real component.
  - Regressing testing may be conducted to ensure that new errors have not been introduced.

■ **Bottom-Up Integration**

- Bottom-up integration testing as its name implies, begins construction and testing with atomic modules.
- A bottom-up integration strategy may be implemented with the following four steps:
  - Low-level components are combined into clusters that perform a specific software subfunction.
  - A driver is written to coordinate test case input and output.
  - The cluster is tested.
  - Drivers are removed and clusters are combined moving upward in the program structure.



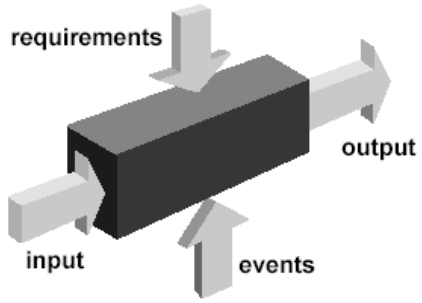
- **Regression testing**

  - Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.
  - The regression test suite contains three different classes of test cases.
    - A representative sample of tests that will exercise all software functions.
    - Additional tests that focus on software functions that are likely to be affected by the change.
    - Tests that focus on the software components that have been changed.

- **Smoke Testing**
  - A common approach for creating "daily builds" for product software.
  - Smoke testing steps:
    - Software components that have been translated into code are integrated into a "build."
      - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
    - A series of tests is designed to expose errors that will keep the build from properly performing its function.
      - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
    - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
      - The integration approach may be top down or bottom up.

11.3 BLACK-BOX TESTING

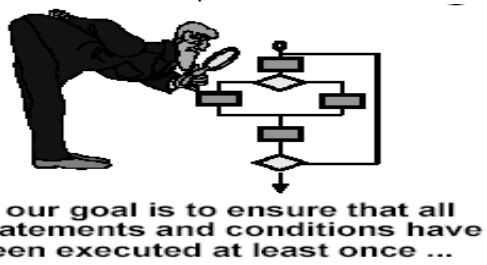
- Black-Box Testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.



- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

11.4 WHITE-BOX TESTING OR GLASS-BOX TESTING

- White-Box Testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by providing test cases that exercise specific sets of conditions and/or loops.



- **Why Cover?**

- Logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- We often believe that a path is not likely to be executed; in fact, reality is often counter intuitive.
- Typographical errors are random; it's likely that untested paths will contain some.

11.5 VALIDATION TESTING

- Focus is on software requirements

  - Validation Test Criteria
  - Configuration review
  - Alpha/Beta testing
    - Focus is on customer usage

11.6 SYSTEM TESTING

- Focus is on system integration
  - Recovery testing
    - forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Security testing
    - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
  - Stress testing
    - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
  - Performance Testing
    - test the run-time performance of software within the context of an integrated system.

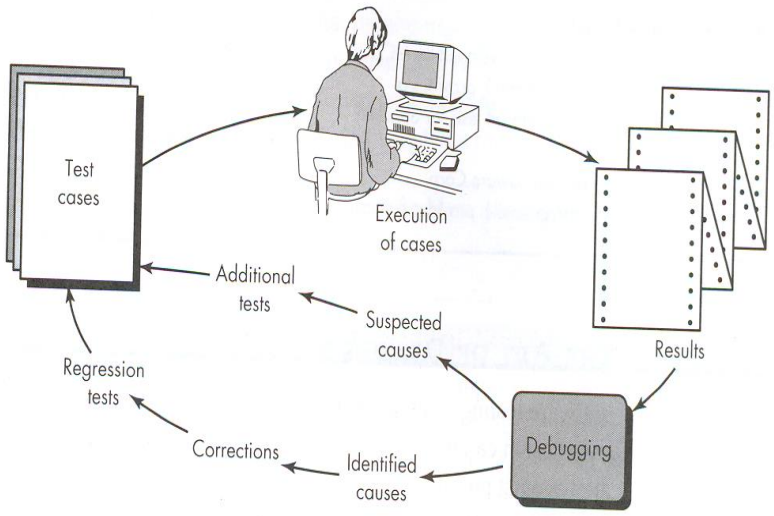
11.7 THE ART OF DEBUGGING

- Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is an action that results in the removal of the error.

- **The Debugging Process:**

  - Debugging will always have one of <u>two</u> outcomes:
    - The cause will be found and corrected
    - The cause will not be found. In the latter case, the person performing debugging may suspect a cause, design one or more test cases to help validate that suspicion, and work toward error correction in an iterative fashion.

- The debugging process



do with an ome clues:

e symptom located at situation.

procession

embedded

systems that couple hardware and software inextricably.

- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

- During debugging, we encounter errors that range from mildly annoying to catastrophic.

■ **Psychological Considerations**

- Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of error increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately corrected.

■ **Debugging Strategies**

- Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition and luck.

- In general, three debugging strategies have been proposed
  - Brute force
  - Backtracking
  - Cause elimination.

- Debugging tactics:
  - The brute force category of debugging is probably the most common and least efficient method of isolating the cause of a software error.

- Backtracking is a fairly common debugging approach that can be used successfully in small programs.
- Cause elimination is manifested by induction or deduction and introduces the concept of binary partitioning.

- Automated debugging

- Each of these debugging approaches can be supplemented with debugging tools that provide semi-automated support for the software engineer as debugging strategies are attempted.

- The people factor

- A fresh viewpoint, unclouded by hours of frustration, can do wonders. A final maxim for debugging might be: when all else fails, get help.

■ **Correcting the error**

- Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck suggests three simple questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

  - Is the cause of the bug reproduced in another part of the program?
  - What "next bug" might be introduced by the fix that I'm about to make?
  - What could we have done to prevent this bug in the first place?

_____

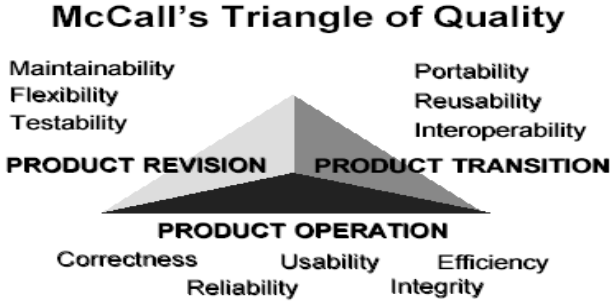
12. PRODUCT METRICS

12.1 SOFTWARE QUALITY

- Software quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- The definition serves to emphasize three important points:
    - Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
    - Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
    - There is asset of implicit requirements that often goes unmentioned. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.
- Software quality is a complex mix of factors that will vary across different applications and the customers who request them.

■ McCall's Quality Factors

- The factors that affect software quality can be categorized in two broad groups:
    - Factors that can be directly measured.
    - Factors that can be measuring only indirectly. In each case measurement should occur. We must compare the software to some datum and arrive at an indication of quality.
- McCall, Richards and Walters propose a useful categorization of factors that affect software quality. These software quality factors, shown in figure, focus on three important aspects of a software product: Its operational characteristics, its ability to undergo change, and its adaptability to new environments.
- Referring to the factors noted in figure, McCall and his colleagues provide the following descriptions:



- *Correctness:* The extent to which a program satisfies its specification and fulfills the customer's mission objectives.
- *Reliability:* The extent to which a program can be expected to perform its intended function with required precision.
- *Efficiency:* The amount of computing resources and code required by a program to perform its function.
- *Integrity:* The extent to which access to software or data by unauthorized persons can be controlled.
- *Usability:* The effort required to learn, operate, prepare input for, and interpret output of a program.
- *Maintainability:* The effort required to locate and fix and error in a program.
- *Flexibility:* The effort required to modify an operational program.
- *Testability:* The effort required to test a program to ensure that it performs its intended function.
- *Portability:* the effort required to transfer the program from one hardware and/or software system environment to another.
- *Reusability:* the extent to which a program can be reused in other applications-related to the packaging and scope of the functions that the program performs.
- *Interoperability:* The effort required to couple one system to another.

A Comment

- McCall's quality factors were proposed in the early 1970s. They are as valid today as they were in that time. It's likely that software built to conform to these factors will exhibit high quality well into the 21st century, even if there are dramatic changes in technology.

■ ISO 9126 Quality Factors:

- The ISO 9126 standard was developed in an attempt to identify quality attributes for computer software. The standard identifies six key quality attributes:
  - *Functionality:* The degree to which the software satisfies stated needs as indicated by the following sub-attributes: suitability, accuracy, interoperability, compliance and security.
  - *Reliability:* The amount of time that the software is available for use as indicated by the following sub-attributes: maturity, fault tolerance, recoverability.
  - *Usability:* the degree to which the software is easy to use as indicated by the following sub-attributes: understandability, learnability, operability.
  - *Efficiency:* The degree to which the software makes optimal use of system resources as indicated by the following sub-attributes: time, behavior, resource behavior.

- *Maintainability:* The ease with which repair may be made to the software as indicated by the following sub-attributes: analyzability, changeability, stability, testability.
- *Portability:* The ease with which the software can be transposed from one environment to another as indicated by the following sub-attributes: adaptability, installability, conformance, replaceabilty.

■ **The Transition to a Quantitative View**

- We examine a set of software metrics that can be applied to the quantitative assessment of software quality. In all cases, the metrics represent indirect measures; that is, we never really measure quality but rather some manifestation of quality. The complicating factor is the precise relationship between the variable that is measuring and the quality of software.

12.2 METRICS FOR THE ANALYSIS MODEL

- *Function-based metrics:* use the function point as a normalizing factor or as a measure of the "size" of the specification
- *Specification metrics:* used as an indication of quality by measuring number of requirements by type

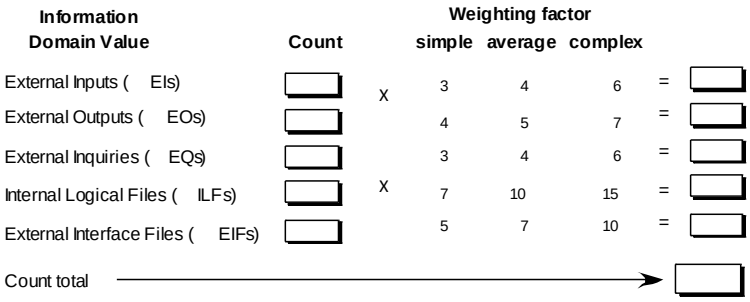
■ **Function-Based Metrics**

- The *function point metric* (FP), first proposed by Albrecht, can be used effectively as a means for measuring the functionality delivered by a system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity
- Information domain values are defined in the following manner:
  - *Number of external inputs (EIs)*
  - *Number of external outputs (EOs)*
  - *Number of external inquiries (EQs)*
  - *Number of internal logical files (ILFs)*
  - *Number of external interface files (EIFs)*

- **Function Points**
  - To compute function points (FP), the following relationship is used:

    $$FP = count\ total\ X\ [0.65 + 0.01\ X \sum (F_i)] \qquad (1)$$

    Where count total is the sum of all FP entries obtained from figure.



- The F_i (i= 1 to 14) are *value adjustment factors* based on responses to the following questions:

    1. Does the system require reliable backup and recovery?
    2. Are specialized data communications required to transfer information to or form the application?
    3. Are there distributed processing functions?
    4. Is performance critical?
    5. Will the system run in an existing, heavily utilized operational environment?
    6. Does the system require on-line data entry?
    7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
    8. Are the ILFs updated on-line?
    9. Are the inputs, outputs, files or inquiries complex?
    10. Is the internal processing complex?
    11. Is the code designed to be reusable?
    12. Are conversion and installation included in the design?
    13. Is the system designed for multiple installations in different organizations?
    14. Is the application designed to facilitate change and for ease of use by the user?

    Each of these questions is answered using a scale that ranges from **0 to 5**.

    The constant values in *equation-1* and the weighting factors that are applied to information domain counts are determined empirically.

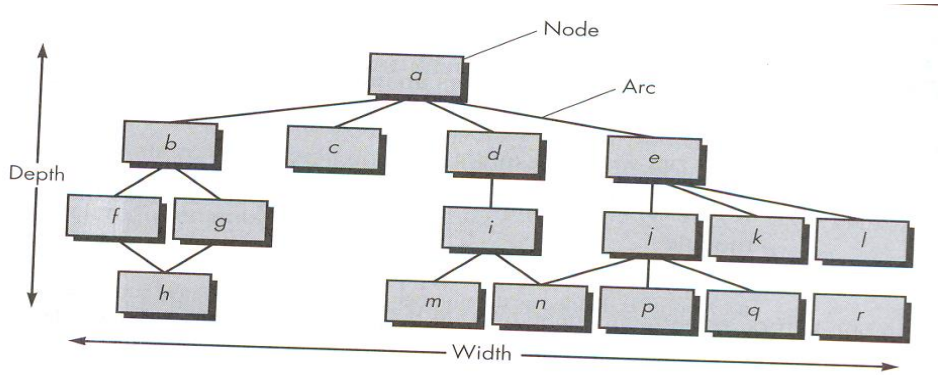
12.3 METRICS FOR THE DESIGN MODEL

- Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied.

■ Architectural Design Metrics

- Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture.

- Architectural design metrics
  - Structural complexity = g(fan-out)
  - Data complexity = f(input & output variables, fan-out)
  - System complexity = h(structural & data complexity)
- HK metric: architectural complexity as a function of fan-in and fan-out
- Morphology metrics: a function of the number of modules and the number of interfaces between modules



- For hierarchical architectures structural complexity of a module I is defined in the following manner:
  - $S(i) = f^2_{out}(i)$

    Where $f_{out}(i)$ is the fan-out of module i.

  - Data complexity provides an indication of the complexity in the internal interface for a module I and is defined as

    $D(i) = v(i) / [f_{out}(i) + 1]$

    Where v(i) is the number of input and output variables that are passed to and from module i.

    Finally, system complexity is defined as the sum of structural and data complexity specified as

    $$C(i) = S(i) + D(i)$$

- As each of these complexity values increases, the overall architectural complexity or the system also increases. This leads to a greater likelihood that integration and testing effort will also increase.

- Fenton suggests a number of simple morphology metrics that enable different program architectures to be compared using a set of straight forward dimensions. Referring to the call-and-return architecture in figure. The following metric can be defined:

  size = n + a

  where n is the number of nodes and a is the number of arcs.

- The U.S. Air Force Systems command has developed a number of software quality indicators that are based on measurable design characteristics of a computer program. The Air Force uses information obtained form data and architectural design to derive a design structure quality index that ranges from 0 to 1. The following values must be ascertained to compute the DSQI.

  S_1 = the total number of modules defined in the program architecture

  S_2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere.

  S_3 = the number of modules whose correct function depends on prior processing.

  S_4 = the number of database items.

  S_5 = the total number of unique database items.

  S_6 = the number of database segments

  S_7 = the number of modules with a single entry and exit.

- *Program Structure*: D_1, where D_1 is defined as follows: If the architectural design was developed using a distinct method, then D_1= 1, otherwise D_1=0.

  Module independence: $D_2 = 1 - (S_2/S_1)$

  Modules not dependent on prior processing: $D_3 = 1 - (S_3/S_1)$

  Database size: $D_4 = 1 - (S_5/S_4)$

  Database compartmentalization: $D_5 = 1 - (S_6/S_4)$

  Module entrance/exit characteristic: $D_6 = 1 - (S_7/S_1)$

  With these intermediate values determined, the DSQI is computed in the following manner:

  DSQI = $\sum w_i D_i$

Where i = 1 to 6, w_i is the relative weighting of the importance of each of the intermediate values, and $\sum w_i = 1$

Metrics for Object-Oriented Design

- Whitmire describes nine distinct and measurable characteristics of an OO design:
    - Size
        - Size is defined in terms of four views: population, volume, length, and functionality
    - Complexity
        - How classes of an OO design are interrelated to one another
    - Coupling
        - The physical connections between elements of the OO design
    - Sufficiency
        - "the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application."
    - Completeness
        - An indirect implication about the degree to which the abstraction or design component can be reused
    - Cohesion
        - The degree to which all operations working together to achieve a single, well-defined purpose
    - Primitiveness
        - Applied to both operations and classes, the degree to which an operation is atomic
    - Similarity
        - The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose
    - Volatility
        - Measures the likelihood that a change will occur

Class-Oriented Metrics

The class is the fundamental unit of an object oriented system.

- *Proposed by Chidamber and Kemerer Metrics Suite*
    - weighted methods per class
    - depth of the inheritance tree
    - number of children
    - coupling between object classes
    - response for a class

- lack of cohesion in methods

- *The MOOD Metrics Suite*
  - Method inheritance factor:
  - Coupling factor
  - Polymorphism factor

- *Proposed by Lorenz and Kidd:*
  - class size
  - number of operations overridden by a subclass
  - number of operations added by a subclass
  - specialization index

■ **Component-Level Design Metrics**

- *Cohesion metrics:* a function of data objects and the locus of their definition
- *Coupling metrics:* a function of input and output parameters, global variables, and modules called
- *Complexity metrics:* hundreds have been proposed (e.g., cyclomatic complexity)

■ **Operation-Oriented Metrics**

- *Proposed by Lorenz and Kidd:*
  - average operation size
  - operation complexity
  - average number of parameters per operation

■ **User Interface Design Metrics**

- *Layout appropriateness:* a function of layout entities, the geographic position and the "cost" of making transitions among entities.

12.4 METRICS FOR SOURCE CODE

- Also called Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program

  - It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed. The measures are

- n_1 = the number of distinct operators that appear in a program.
- n_2 = the number of distinct operands that appear in a program.
- N_1 = the total number of operator occurrences.
- N_2 = the total number of operand occurrences.
- Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined.

$$V = N \log_2 (n_1+n_2)$$

- It should be noted that V will vary with programming language and represents the volume of information required to specify a program.
- Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program.

  L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

$$L = 2/n_1 \text{ X } n_2/N_2$$

12.5 METRICS FOR TESTING

- Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower. For this reason, the tester should expand above average effort to uncover errors in such modules before they are integrated in a system.

■ Halstead Metrics Applied to Testing

- Testing effort can also be estimated using metrics derived from Halstead measures
  - Using the definitions for program volume, V, and program level, PL, Halstead effort, e, can be computed as

    $PL = 1 / [n_1/2) \text{ X } (N_2/n_2)]$ and $e = V/PL$

- The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship: Percentage of testing effort $(k) = e(k) / \sum e(i)$
  Where $e(k)$ is computed for module k using equations and the summation in the denominator of equation is the sum of Halstead effort across all modules of the system.

■ Metrics for Object-Oriented Testing

- Binder suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system.
  - *Lack of cohesion in methods (LCOM):* The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

- *Percent public and protected (PAP)*: This metric indicates the percentage of class attributes that are public or protected.

- *Public access to data members (PAD)*: This metric indicates the number of classes that can be access another class's attributes, a violation of encapsulation.

- *Number of root classes (NOR)*: This metric is a count of the distinct class hierarchies that are described in the design model.

- *Fan-in (FIN)*: When used in the OO context, fan-in for the inheritance hierarchy is an indication of multiple inheritance. FIN>1 indicates that a class inherits its attributes and operations from more than one root class.

- *Number of children (NOC) and depth of the inheritance tree (DIT)*: Superclass methods will have to be retested for each subclass.

12.6 METRICS FOR MAINTENANCE

- IEEE Std. 982.1 – 1988 suggests a software maturity index that provides an indication of the stability of a software product. The following information is determined:

  M_T = the number of modules in the current release.

  F_c = the number of modules in the current release that have been changed.

  F_a = the number of modules in the current release that have been added.

  F_d = the number of modules from the preceding release that were deleted in the current release.

  The software maturity index is computed in the following manner:

  $$SMI = [M_r – (F_a + F_c + F_d)]/M_T$$

  As SMI approaches 1.0, the product begins to stabilize. SMI may also be used as a metric for planning software maintenance activities. The mean time to produce a release of a software product can be correlated with SMI, and empirical models for maintenance effort can be developed.

13. METRICS FOR PROCESS & PROJECTS

- Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework.

13.1 SOFTWARE MEASUREMENT

- Software measurement can be categorized in two ways.
  - *Direct measures* of the software process and product.
  - *Indirect measures* of the product that includes functionality, quality, complexity, efficiency, reliability, maintainability.

■ **Size-Oriented Metrics**
- errors per KLOC (thousand lines of code)
- defects per KLOC
- $ per LOC
- pages of documentation per KLOC
- errors per person-month
- Errors per review hour
- LOC per person-month
- $ per page of documentation

■ **Function-Oriented Metrics**
- errors per FP (thousand lines of code)
- defects per FP
- $ per FP
- pages of documentation per FP
- FP per person-month

■ **Reconciling LOC and FP Metrics**

- The relationship between lines of code and function points depend upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures.

- The following table provides rough estimates of the average number of lines of code required to build one function point in various programming languages.

Programming Language	LOC per Function point avg.	median	low	high
Ada	154	-	104	205
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
COBOL	77	77	14	400
Java	63	53	77	-
JavaScript	58	63	42	75
Perl	60	-	-	-
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
Visual Basic	47	42	16	158

- **Why Opt for FP?**

  - Programming language independent.
  - Used readily countable characteristics that are determined early in the software process.
  - Does not "penalize" inventive (short) implementations that use fewer LOC that other clumsier version.
  - Makes it easier to measure the impact of reusable components.

- **Object-Oriented Metrics**

  - *Number of scenario scripts (use-cases):* A Scenario script is a detailed sequence of steps that describes the interaction between the user and the application.

  - *Number of Key classes:* Key classes are the "highly independent components" that are defined early in object-oriented analysis.

  - *Number of support classes:* Supports Classes are required to implement the system but are not immediately related to the problem domain.

  - *Average number of support classes per key class (analysis class):* The average number of support classes per key class were known for a given problem domain, estimating would be much simplified.

  - *Number of subsystems:* A subsystem is an aggregation of classes that support a function that is visible to the end-user of a system.

- **Use-Case Oriented Metrics**

  - A normalization measure similar to LOC and FP.
  - Used for estimation before significant modeling and construction activities.
  - Independent of programming languages.
  - No standard size for use-case.

■ **Web Engineering Project Metrics**

- Number of static Web pages (the end-user has no control over the content displayed on the page)
- Number of dynamic Web pages (end-user actions result in customized content displayed on the page)
- Number of internal page links (internal page links are pointers that provide a hyperlink to some other Web page within the WebApp)
- Number of persistent data objects
- Number of external systems interfaced
- Number of static content objects
- Number of dynamic content objects
- Number of executable functions

13.2 METRICS FOR SOFTWARE QUALITY

- The overriding goal of software engineering is to produce a high quality system, application, or product within a timeframe that satisfies a market need. To achieve this goal:
  - Software Engineer must apply effective methods coupled with modern tools.
  - Software Engineer must measure a high quality is to be realized.
  - Private metrics collected are assimilated to provide project level results.
  - This metrics provide the effectiveness of individual and group software quality assurance and control activities.
  - Error data can also be used to compute defect removal efficiency for each process framework activity.

■ **Measuring Quality**

- *Correctness* — the degree to which a program operates according to specification
- *Maintainability*—the degree to which a program is amenable to change
- *Integrity*—the degree to which a program is impervious to outside attack
- *Usability*—the degree to which a program is easy to use

■ **Defect Removal Efficiency (DRE)**

- A quality metric that provides benefits at both the project and process level is defect removal efficiency.

- When considered for a project as a whole, DRE is defined in the following manner:

$$DRE = E / (E + D)$$

Where E is the number of errors found before delivery of the software to the end-user

D is the number of defects found after delivery.

_____

14. RISK MANAGEMENT

Project Risks

What can go wrong?

What is the likelihood?

What will the damage be?

What can we do about it?

14.1 REACTIVE vs PROACTIVE RISK STRATEGIES

■ **Reactive Risk Management**

- project team reacts to risks when they occur
- mitigation—plan for additional resources in anticipation of fire fighting
- fix on failure—resource are found and applied when the risk strikes
- crisis management—failure does not respond to applied resources and project is in jeopardy

■ **Proactive Risk Management**

- formal risk analysis is performed
- organization corrects the root causes of risk
    - TQM concepts and statistical SQA
    - examining risk sources that lie beyond the bounds of the software
    - developing the skill to manage change

14.2 SOFTWARE RISKS

- Risk always involves two characteristics
    - Uncertainty : the risk may or may not happen; that is, there are no 100% probable risks.
    - Loss : if the risk becomes a reality, unwanted consequences or losses will occur.

- *Project risks* threaten the project plan. Project risks identify potential budgetary, schedule, personal, resource, stakeholder, and requirements problems and their impact on a software project.

- *Technical risks* threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks occur because the problem is harder to solve than we thought it would be.

- *Business risks* threaten the viability of the software to be built.

  - The top five business risks are:
    (1) Building an excellent product or system that no one really ants,

    (2) Building a product that no longer fits into the overall business strategy for the company,

    (3) Building a product that the sales force doesn't understand how to sell,

    (4) Losing the support of senior management due to a change in focus or a change in people and

    (5) Losing budgetary or personnel commitment.

- Predictable risks are extrapolated from past project experience.

- Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

Seven principles of risk management

1. *Maintain a global perspective*—view software risks within the context of system and the business problem
2. *Take a forward-looking view*—think about the risks that may arise in the future; establish contingency plans
3. *Encourage open communication*—if someone states a potential risk, don't discount it.
4. *Integrate*—a consideration of risk must be integrated into the software process
5. *Emphasize a continuous process*—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.
6. *Develop a shared product vision*—if all stakeholders share the same vision of the software, it likely that better risk identification and assessment will occur.
7. *Encourage teamwork*—the talents, skills and knowledge of all stakeholder should be pooled

14.3. RISK IDENTIFICATION

- *Product size*—risks associated with the overall size of the software to be built or modified.

- *Business impact*—risks associated with constraints imposed by management or the marketplace.

- *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.

- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.

- *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

- *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

■ Assessing Project Risk

- Have top software and customer managers formally committed to support the project?
- Are end-users enthusiastically committed to the project and the system/product to be built?
- Are requirements fully understood by the software engineering team and their customers?
- Have customers been involved fully in the definition of requirements?
- Do end-users have realistic expectations?
- Is project scope stable?
- Does the software engineering team have the right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?
- Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

■ Risk Components

- *performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.

- *cost risk*—the degree of uncertainty that the project budget will be maintained.
- *support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

14.4 RISK PROJECTION

- *Risk projection*, also called *risk estimation,* attempts to rate each risk in two ways
  - the likelihood or probability that the risk is real
  - the consequences of the problems associated with the risk, should it occur.
- The are four risk projection steps:
  - establish a scale that reflects the perceived likelihood of a risk
  - delineate the consequences of the risk
  - estimate the impact of the risk on the project and the product,
  - note the overall accuracy of the risk projection so that there will be no misunderstandings.

■ **Developing a Risk Table**

Risk	Probability	Impact	RMMM
			Risk Mitigation Monitoring & Management

- Estimate the probability of occurrence
- Estimate the impact on the project on a scale of 1 to 5, where
  - 1 = low impact on project success
  - 5 = catastrophic impact on project success
- sort the table by probability and impact

■ **Assessing Risk Impact**

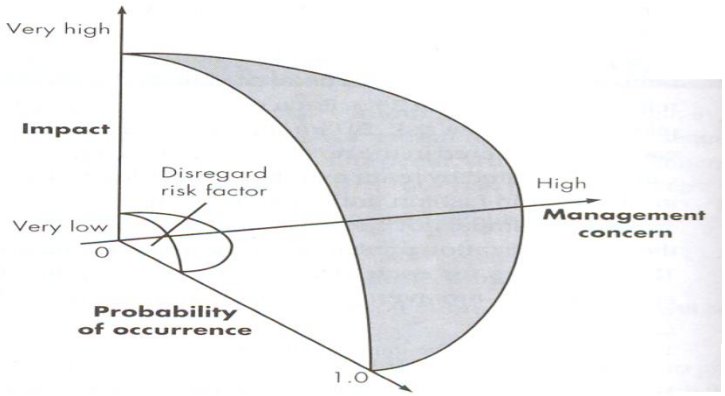
- The overall *risk exposure,* RE, is determined using the following relationship:

RE = *P* x *C*

where

P is the probability of occurrence for a risk, and

C is the cost to the project should the risk occur.



14.5 RISK REFINEMENT

- During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

- One way to do this is to represent the risk in condition transition-consequence format. That is, the risk is stated in the following form:

    - Given that <condition> then there is concern that <consequence>.
- This general condition can be refined in the following manner:

    - *Subcondition1:* Certain reusable components were developed by a third party will no knowledge of internal design standards.

    - *Subcondition2:* The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

    - *Subcondition3:* Certain reusable components have been implemented in a language that is not supported on the target environment.

14.6 RISK MITIGATION, MONITORING, AND MANAGEMENT

- Mitigation—how can we avoid the risk?
- Monitoring—what factors can we track that will enable us to determine if the risk is becoming more or less likely?
- Management—what contingency plans do we have if the risk becomes a reality?

14.7 THE RMMM PLAN

- The RMMM plan documents all work performed as part of risk analysis and are used by the project manager as part of the overall project plan.
- Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a Risk Information Sheet (RIS).
- In most cases, the RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.
- Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence.
- Risk monitoring is a project tracking activity with three primary objectives:
    1. To assess whether predicted risks do, in fact, occur.
    2. To ensure that risk aversion steps defined for the risk are being properly applied.
    3. To collect information that can be used for future risk analysis.

Risk Information Sheet

Project: Embedded software for XYZ system.

Risk type: schedule risk

Priority (1 low ... 5 critical): 4

Risk factor: Project completion will depend on tests which require hardware component under development. Hardware component delivery may be delayed.

Probability: 60 %

Impact: Project completion will be delayed for each day that hardware is unavailable for use in software testing.

Monitoring approach:

Scheduled milestone reviews with hardware group.

Contingency plan:

Modification of testing strategy to accommodate delay using software simulation.

Estimated resources: 6 additional person months beginning 7-1-96

———

15. QUALITY MANAGEMENT

15.1 QUALITY CONCEPTS

- Variation control is the heart of the quality control. A manufacturer wants to minimize the variation among the products that are produced.

■ Quality

- The *American Heritage Dictionary* defines *quality* as
  - "a characteristic or attribute of something."

- For software, two kinds of quality may be encountered:
  - Quality of design encompasses requirements, specifications, and the design of the system.
  - Quality of conformance is an issue focused primarily on implementation.
  - user satisfaction = compliant product + good quality + delivery within budget and schedule

■ Quality Control

- Quality control involves the series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it.
- Quality control includes a feedback loop to the process that created the work product.
- A key concept of quality control is that all work products have defined, measurable specifications to which h we may compare the output of each process.
- The feedback loop is essential to minimize the defects produced.

■ Quality Assurance

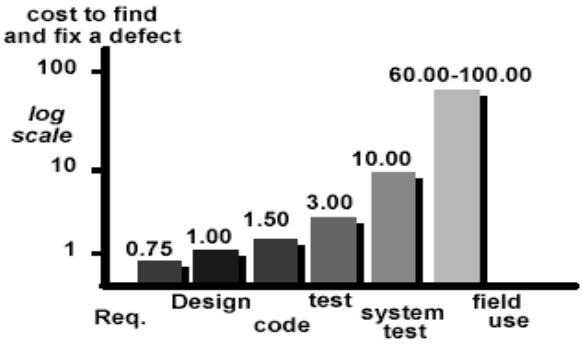
- Quality assurance consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control activities.
- The goal of quality assurance is to provide management with the data necessary to be informed about product quality, thereby gaining insight and confidence that product quality is meeting its goals.

■ Cost of Quality

- *Prevention costs* include
  - quality planning
  - formal technical reviews
  - test equipment
  - Training

- *Internal failure costs* include
  - rework
  - repair
  - failure mode analysis

- *External failure costs* are
  - complaint resolution
  - product return and replacement
  - help line support
  - warranty work



15.2 SOFTWARE QUALITY ASSURANCE

- Software Quality can be defined as Conformance to explicitly state functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.
- Definition serves to emphasize three important points:

    - Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
    - Specified standard define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
    - A set of implicit requirements often goes unmentioned. If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect.

■ SQA Activities

- Prepares an SQA plan for a project.
    - The plan identifies

- Evaluations to be performed.
- Audits and reviews to be performed.
- Standards that is applicable to the project.
- Procedures for error reporting and tracking.
- Documents to be produced by the SQA group.
- Amount of feedback provided to the software project team.

- Participates in the development of the project's software process description.

- The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

- Reviews software engineering activities to verify compliance with the defined software process.
  - Identifies, documents, and tracks deviations from the process and verifies that corrections have been made.

- Audits designated software work products to verify compliance with those defined as part of the software process.
  - Reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made
  - Periodically reports the results of its work to the project manager.

- Ensures that deviations in software work and work products are documented and handled according to a documented procedure.

- Records any noncompliance and reports to senior management.
  - Noncompliance items are tracked until they are resolved.

15.3 SOFTWARE REVIEWS

- Software reviews are a "filter" for the software process. That is, reviews are applied at various points during software engineering and serve to uncover errors and defects that can then be removed.

- **What Are Reviews?**

  - a meeting conducted by technical people for technical people
  - a technical assessment of a work product created during the software engineering process

- a software quality assurance mechanism
- a training ground

- **What Reviews Are Not?**

  - A project summary or progress assessment
  - A meeting intended solely to impart information
  - A mechanism for political or personal reprisal!

- **Cost impact of software defects**

  - The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.

  - The obvious benefit of formal technical reviews is the early discovery of errors so that they do not propagate to the next step in the software process.

  - To illustrate the cost impact of early error detection, we consider a series of relative costs that are based on actual cost data collected for large software projects.

- **Defect Amplification and Removal**

  - A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of a software engineering process.

  - During the step, errors may be inadvertently generated. Review may fail to uncover newly generated errors and errors from previous steps, resulting in some number of errors that are passed through.

  - To conduct reviews, a software engineer must expend time and effort, and the development organization must spend money.

15.4 FORMAL TECHNICAL REVIEWS

- A formal technical review is a software quality control activity performed by software engineers.

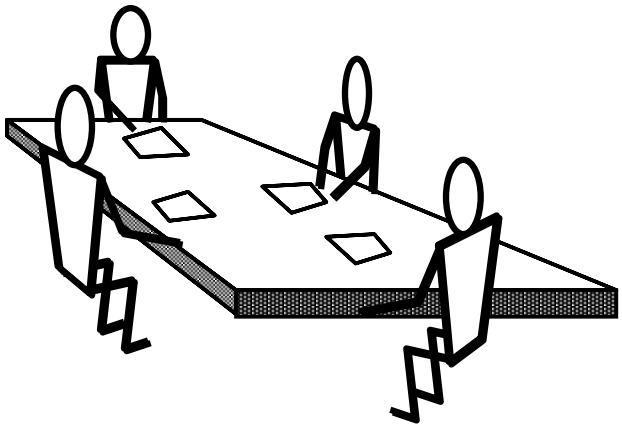
- The objectives of formal technical reviews are:

1. to uncover errors in function, logic, or implementation for any representation of the software.
2. to verify that the software under review meets its requirements.
3. to ensure that the software has been represented according to predefined standards.
4. to achieve software that is developed in a uniform manner
5. to make projects more manageable.

■ The Review Meeting

- Every review meeting should abide by the following constraints:

1. Between three and five people should be involved in the review.
2. Advance preparation should occur but should require no more than two hours of work for each person.
3. The duration of the review meeting should be less than two hours.



- be prepared—evaluate product before the review
- review the product, not the producer
- keep your tone mild, ask questions instead of making accusations
- stick to the review agenda
- raise issues, don't resolve them
- avoid discussions of style—stick to technical correctness
- schedule reviews as project tasks
- record and report all review results

■ Review Reporting and Record Keeping

- A review summary report answers three questions:

1. What was reviewed?

2. Who reviewed it?

3. What were the findings and conclusions?

- The review summary report is a single page form.

- The review issues list serves two purposes:

    1. To identify problem areas within the product
    2. To serve as an action item checklist that guides the producer as corrections are made. An issues list is normally attached to the summary report.

- It is important to establish a follow-up procedure to ensure that items on the issues list have been properly corrected.

■ Review Guidelines

- The following represents a minimum set of guidelines for formal technical reviews:

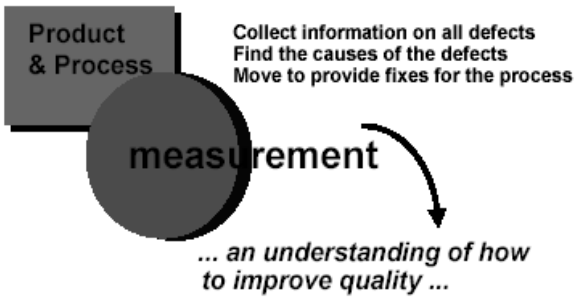
    1. Review the product, not the producer
    2. Set an agenda and maintain it
    3. Limit debate and rebuttal
    4. Enunciate problem areas
    5. Take written notes
    6. Limit the number of participants and insist upon advance preparation
    7. Develop a checklist for each product that is likely to be reviewed.
    8. Allocate resources and schedule time for FTRs
    9. Conduct meaningful training for all reviewers.
    10. Review your early reviews.

■ Sample-Driven Reviews (SDRs)

- SDRs attempt to quantify those work products that are primary targets for full FTRs.
- *To accomplish this …*
    - Inspect a fraction ai of each software work product, *i.* Record the number of faults, fi found within ai.
    - Develop a gross estimate of the number of faults within work product *i* by multiplying fi by 1/ai.
    - Sort the work products in descending order according to the gross estimate of the number of faults in each.
    - Focus available review resources on those work products that have the highest estimated number of faults.

15.5 STATISTICAL SOFTWARE QUALITY ASSURANCE

- The software statistical quality assurance implies the following steps:
    1. Information about software defects is collected and categorized.
    2. An attempt is made to trace each defect to its underlying cause.
    3. Using the Pareto principle (80% of the defects can be traced to 20% of all possible causes), isolate the 20% (the "vital few").
    4. Once the vital few causes have been identified, move to correct the problems that have caused the defects.



■ Six-Sigma for Software Engineering

- The term "six sigma" is derived from six standard deviations—3.4 instances (defects) per million occurrences—implying an extremely high quality standard.

- The Six Sigma methodology defines three core steps:

    - *Define* customer requirements and deliverables and project goals via well-defined methods of customer communication
    - *Measure* the existing process and its output to determine current quality performance (collect defect metrics)
    - *Analyze* defect metrics and determine the vital few causes.

- If an existing software process is in place, but improvement is required, Six Sigma suggests two additional steps:

    - *Improve* the process by eliminating the root causes of defects.
    - *Control* the process to ensure that future work does not reintroduce the causes of defects.

- If an organization is developing a software process the core steps are augmented as follows:

    - *Design* the process to (1) avoid the root causes of defects and (2) to meet customer requirements.
    - *Verify* that the process model will, in fact, avoid defects and meet customer requirements.

15.6 SOFTWARE RELIABILITY

- Software reliability is defined in statistical terms as "the probability of failure-free operation of a computer program in a specified environment for a specified time.

■ Measures of reliability and Availability

- A simple measure of reliability is *mean-time-between-failure* (MTBF), where
  MTBF = MTTF + MTTR

- The acronyms MTTF and MTTR are *mean-time-to-failure* and *mean-time-to-repair*, respectively.

- *Software availability* is the probability that a program is operating according to requirements at a given point in time and is defined as
  Availability = [MTTF/(MTTF + MTTR)] x 100%

■ Software Safety

- *Software safety* is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.

- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

- A modeling and analysis is conducted as part of software safety. Initially, hazards are identified and categorized by criticality and risk. For example, some of the hazards associated with a computer-based cruise control for an automobile might be.

  - Causes uncontrolled acceleration that cannot be stopped.
  - Does not respond to depression of brake pedal.
  - Does not engage when switch is activated.
  - Slowly loses or gains speed.

15.7 THE ISO 9000 QULAITY STANDARDS

- A quality assurance system may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

- Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications.

- ISO 9000 describes a quality assurance system in generic terms that can be applied to any business regardless of the products or services offered.

The ISO 9001:2000 Standard

- ISO 9001:2000 is the quality assurance standard that applies to software engineering.

- The standard contains 20 requirements that must be present for an effective quality assurance system.

- The requirements delineated by ISO 9001:2000 address topics such as

    - Management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.